



iSeries
WebSphere® Development Studio
ILE C/C++ Programmer's Guide

Version 5

SC09-2712-03





@server

iSeries

WebSphere[®] Development Studio

ILE C/C++ Programmer's Guide

Version 5

SC09-2712-03

Note!

Before using this information and the product it supports, be sure to read the general information under “Notices” on page 477.

Third Edition (May 2001)

This edition applies to Version 5, Release 1, Modification 0, of IBM WebSphere Development Studio for iSeries (program 5722-WDS), ILE C/C++ compilers, and to all subsequent releases and modifications until otherwise indicated in new editions. This edition replaces ILE C for AS/400 Programmer’s Guide (SC09–2712–01) and ILE C++ for AS/400 Programming Guide (SC09–2417–01).

Changes or additions to the text and illustrations are indicated by a vertical line to the left of the change or addition.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

IBM welcomes your comments. You can send comments to:

IBM Canada Ltd. Laboratory,
2G/KB7/1150/TOR
1150 Eglinton Avenue East
Toronto, Ontario, Canada. M3C 1H7

You can also send your comments by facsimile (attention: RCF Coordinator), or you can send your comments electronically to IBM. See “How to Send Your Comments” on page x for a description of the methods.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1993, 2002. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

About This Guide	ix
Who Should Use This Guide	ix
Prerequisite and Related Information	ix
Install Licensed Program Information	x
A Note About Examples	x
Control Language Commands	x
How to Send Your Comments	x

Figures	xiii
--------------------------	-------------

Tables	xvii
-------------------------	-------------

Part 1. Introduction 1

Chapter 1. Introduction to the ILE C/C++ Compiler 3

ILE and OS/400 Operating System Programming Features	3
Programming Languages Supported by the OS/400 Operating System	3
Program Creation	3
Program and Resource Management	4
Program Calls	5
Program Debugging	5
Bindable APIs	5
Main Features of the C++ Language	5
Using C++ for Object-Oriented Programming	6
Coding a C++ Program	6

Part 2. Creating and Compiling Programs 17

Chapter 2. Creating a Program 19

Introducing the Program Development Process	19
Preparing a Program	19
Compiling	19
Binding	20
Running	20
Entering Source Statements	20
Creating a Program in One Step	22
Creating a Program in Two Steps	24
Identifying Program and User Entry Procedures	24
Understanding the Internal Structure of a Program Object	25
Using Static Procedure Calls	25
Working with Binding Directories	25
Using the Binder to Create a Program	26
Preparing to Create a Program	26
Specifying Parameters for the CRTPGM Command	27
Resolving Import Requests	27
Using a Binder Listing	28
Updating a Module or a Program Object	29

Activation Groups	30
-----------------------------	----

Chapter 3. Service Programs 33

Differences between Programs and Service Programs	33
Public Interface	33
Using the Binder to Create a Service Program	34
Considerations when Creating a Service Program	34
Specifying Parameters for the CRTSRVPGM Command	34
Updating or Changing a Service Program	34
Using Related CL commands	35
Creating a Sample Service Program	35
Creating the Source Files	35
Compiling and Binding the Service Program	37
Binding the Service Program to a Program	37

Chapter 4. Working With Exports From Service Programs 39

Determining Exports from Service Programs	39
Displaying Export Symbols With the Display Module Command	39
Creating a Binder Language Source File	40
Creating Binder Language Using SEU	40
Creating Binder Language Using the RTVBNDSRC Command	41
Updating a Service Program Export List	42
Using the Demangling Functions	42
Handling Unresolved Import Requests During Program Creation	43
Creating a Service Program Using Binder Language	44
Creating a Program with Circular References	44
Creating the Source Files	45
Creating Modules	46
Creating Binder Language	46
Creating the Program	47
Handling Unresolved Import Requests with *UNRSLVREF	47
Handling Unresolved Import Requests by Changing Program Creation Order	48
Binding a Program to a Non-Existent Service Program	49
Updating a Service Program Export List	50
Program Description	50
Creating the Source Files	51
Compiling and Binding Programs and Service Programs	52
Running the Program	53

Chapter 5. Example - Creating a Sample ILE C Application 55

Sample Application Overview	55
Task Overview	56
Instructions	57

Chapter 6. Running a Program 69

The ILE C/C++ Run-Time Model	69
Activation Groups	70
Calling a Program	71
Calling a Program Using the TFRCTL Command	71
Running a Program from a User-Created CL Command	73
Using the Call Command.	75
Using the CL CALL Command	75
Passing Parameters to a Program	75
Ending a Program	77
Managing Activation Groups	78
Specifying an Activation Group.	78
Presence of a Program on the Call Stack.	81
Deleting an Activation Group	81
Reclaiming System Resources	82
Managing Run-Time Storage.	82
Managing the Default Heap	83
Messaging Support.	84

Part 3. Debugging Programs 85

Chapter 7. Debugging a Program 87

Avoiding Modification of Production Files	87
Debug Limits of the ILE Source Debugger	87
Preparing a Program for Debugging	88
Creating a Listing View	88
Debug Commands	88
Starting a Source Debug Session	90
Adding and Removing Programs from a Debug Session	93
Setting Debug Options	94
Viewing the Program Source.	95
Displaying Other Modules in Your Program	95
Displaying a Different View of a Module	96
Setting and Removing Breakpoints	96
Setting and Removing Unconditional Breakpoints	99
Setting and Removing Conditional Thread Breakpoints	99
Using the Work with Module Breakpoints Display	100
Using the TBREAK or CLEAR Debug Commands	100
Setting a Conditional Breakpoint Using F13	100
Setting a Conditional Breakpoint Using the BREAK Command	100
Removing All Breakpoints	101
Setting and Removing Watch Conditions	101
Characteristics of Watches	101
Setting Watch Conditions	102
Using the WATCH Command.	102
Displaying Active Watches	104
Removing Watch Conditions	104
Example of Setting a Watch Condition	105
Stepping through the Program	106
Stepping over Programs.	106
Stepping into Programs	106
Stepping over Procedures	109
Stepping into Procedures	110
Displaying or Changing the Value of Variables	111

Changing the Value of Scalar Variables	113
Equating a Name with a Variable, Expression, or Command	114
Displaying a Structure	115
Displaying Variables as Hexadecimal Values	115
Displaying Null Ended Character Arrays	116
Displaying Character Arrays	117
Using F11 to Display Variables	118
Sample EVAL Commands for Pointers, Variables, and Bit Fields	119
EVAL Commands for System and Space Pointers	121
Changing Optimization and Observability.	123
Changing Optimization Levels	123
Removing Module Observability	125
Sample Source for EVAL Commands	126
Sample Source for Displaying System and Space Pointers	127
Sample Source for Displaying C++ Constructs	129
ILE Source Debugger and ILE C Application Hints	130
Debug Language Syntax.	137

Part 4. Performing I/O Operations 139

Chapter 8. Using Stream and Record I/O Functions with iSeries Data Management Files 141

Record Files	141
Stream Files	142
Stream Buffering	142
Text Streams and Binary Streams	142
Text Streams.	143
Binary Streams	143
Open Modes for Dynamically Created Stream Files	143
stdin, stdout, and stderr	144
Session Manager	145
iSeries System Files	145
File Naming Conventions	146
Opening Text Stream Files	147
Writing Text Stream Files	148
Reading Text Stream Files	149
Updating Text Stream Files	150
Opening Binary Stream Files (one character at a time)	151
Writing Binary Stream Files (one character at a time)	152
Reading Binary Stream Files (one character at a time)	152
Updating Binary Stream Files (one character at a time)	153
Opening Binary Stream Files (one record at a time)	156
Writing Binary Stream Files (one record at a time)	156
Reading Binary Stream Files (one record at a time)	157
Open Feedback Area	158
I/O Feedback Area	158

Chapter 9. Using ILE C/C++ Stream Functions with the iSeries Integrated File System 161

The Integrated File System	161
Root File System	162
Open Systems File System	162
Library File System	163
Document Library Services File System.	164
LAN Server/400 File System	164
Optical Support File System	165
File Server File System	166
Enabling Integrated File System Stream I/O	167
Using Stream I/O with Large Files	167
Stream Files	167
Stream Files Versus Database Files	167
Text Streams.	168
Binary Streams	169
Opening Text Stream and Binary Stream Files	170
Storing Data as a Text Stream or as a Binary Stream	170
Using the Integrated File System	171
Copying Source Files into the Integrated File System	172
Editing Stream Files	172
The SRCSTMF Parameter	172
Header File Search	172
Preprocessor Output	179
Listing Output	179
Code Pages and CCSIDs.	179
Pitfalls to Avoid	180
Examples of Using Integrated File System Source.	180
Using Stream I/O	181

Part 5. Working with iSeries File Systems and and Devices. 183

Chapter 10. Using Externally Described Files in Your Programs . . . 185

Typedefs	185
Header Description	186
Level Checking.	187
Record Format Name.	188
Record Field Names	189
Alignment of Fields in C Structures	189
Using Externally Described Physical and Logical Database Files	190
Input and Both Fields	190
Key Fields	190
Using Externally Described Device Files	192
Input Fields	192
Output Fields	193
Both Fields	193
Indicator Field	194
Separate Indicator Area	194
Indicator in the File Buffer	196
Using Multiple Record Formats	196
Using Packed Decimal and Zoned Decimal Data	199
Using Long Names for Files	200

Chapter 11. Using Database Files and Distributed Data Management Files In Your Programs. 201

Database Files and Distributed Data Management Files	201
Physical Files and Logical Files	201
Data Files and Source Files	202
Access Paths	203
Arranging Key Fields.	203
Duplicate Key Values.	203
Deleted Records	204
Locking	204
Sharing	204
Null Capable Fields	205
Opening Database and DDM Files as Record Files	206
Record Functions for Database and DDM Files	206
I/O Considerations for DDM Files	206
Opening Database and DDM Files as Binary Stream Files	207
I/O Considerations for Binary Stream Database and DDM Files.	207
Binary Stream Functions for Database and DDM Files	207
Processing a Database Record File in Arrival Sequence	207
Processing a Database Record File in Keyed Sequence	209
Processing a Database Record File Using Record Input and Output Functions	210
Grouping File Operations Using Commitment Control	213
Blocking Records	217

Chapter 12. Using Device Files in Your Programs 219

OS/400 Feedback Areas for all Device Files	219
Display Files, Intersystem Communication Files and Printer Files	219
I/O Considerations	219
Separate Indicator Areas.	219
Major and Minor Return Codes	220
Display Files and Subfiles	220
I/O Considerations for Display Files	220
I/O Considerations for Subfiles	221
Opening Display Files and Subfiles as Binary Stream Files	221
I/O Considerations for Binary Stream Subfiles	221
Program Devices for Binary Stream Display Files	221
Binary Stream Functions for Display Files and Subfiles	221
Open Display Files as Record Files	221
I/O Considerations for Record Display Files	222
I/O Considerations for Record Subfiles.	222
Record Functions for Display Files and Subfiles	222
Specifying Indicators as Part of the File Buffer	223
Specifying Indicators in a Separate Indicator Area	224
Establishing the Default Program Device	226
Changing the Default Program Device	228
Using Feedback Information	231

Using Subfiles	233
Using Intersystem Communication Function Files	236
I/O Considerations for Intersystem Communication Function Files	236
Opening ICF Files as Binary Stream Files	236
I/O Considerations for Binary Stream ICF Files	236
Binary Stream Functions for ICF Files	236
Opening ICF Files as Record Files	237
I/O Considerations for Record ICF Files	237
Record Functions for ICF Files.	238
Using Printer Files.	242
I/O Considerations for Printer Files	243
Opening Printer Files as Binary Stream Files	243
Opening Printer Files as Record Files	243
Record Functions for Printer Files	243
Writing to a Tape File	245
I/O Considerations for Tape Files	245
Opening Tape Files as Binary Stream Files.	246
Binary Stream Functions for Tape Files	246
Opening Tape Files as Record Files	246
Record Functions for Tape Files	247
Writing to a Diskette File	249
I/O Considerations for Diskette Files	249
Opening Diskette Files as Binary Stream Files	250
Binary Stream Functions for Diskette Files.	250
Opening Diskette Files as Record Files	250
Record Functions for Diskette Files	250
Using Save Files	252
I/O Considerations for Save Files	252
Opening Save Files as Binary Stream Files.	253
I/O Considerations for Binary Stream Save Files	253
Binary Stream Functions for Save Files	253
Opening Save Files as Record Files	253
I/O Considerations for Record Save Files	253
Record Functions for Save Files	253

Part 6. Working with iSeries Features 255

Chapter 13. Handling Exceptions in Your Program 257

Handling Exceptions	259
Checking the Return Value of a Function	259
Checking the Erno Value	260
Checking the Global Variable _EXCP_MSGID	261
Checking the System Exceptions for Stream Files	261
Checking the System Exceptions for Record Files	261
Using Exception Handlers	265
Unhandled Exceptions	267
Using Direct Monitor Handlers	268
Exception Classes	269
Control Actions.	270
Specifying Message Identifiers.	271
Nested Exceptions.	278
Using Cancel Handlers	278
Using Integrated Language Environment Condition Handlers	281
Using the C/C++ signal Function to Handle Exceptions	287

Control Boundary Examples for ILE C/C++	292
Exception Percolation: an Example	296

Chapter 14. Using iSeries Pointers in Your Program 299

Using Open Pointers	300
Using Pointers Other Than Open Pointers.	300
Declaring Pointer Variables.	300
Using Pointer Casting	303

Chapter 15. Using Packed Decimal Data in Your C Programs 309

Converting from Packed Decimal Data Types.	309
Converting from a Packed Decimal Type to a Packed Decimal Type.	309
Converting from a Packed Decimal Type to an Integer Type.	311
Converting from a Packed Decimal Type to a Floating Point Type	312
Overflow Behavior	313
Passing Packed Decimal Data to a Function	313
Passing a Pointer to a Packed Decimal Variable to a Function	314
Calling Another Program that Contains Packed Decimal Data	315
Using Library Functions with a Packed Decimal Data Type	316
Understanding Packed Decimal Data Type Errors	320

Chapter 16. Calling Conventions . . . 325

Program and Procedure Calls	325
Calling Programs	325
Calling Conventions for Dynamic Program Calls	326
Calling Procedures for ILE C	345
Calling C++ Programs and Procedures from ILE C	351
Calling Procedures for ILE C++	351
Introducing the Call Stack	352
Calling a Program Using a Linkage Specification	353
Calling an ILE Procedure Using a Linkage Specification.	354
Passing Parameters	355
Passing Parameters in C++.	355
Using Default Parameter Passing Styles	359
Using Operational Descriptors.	360
Understanding Data-Type Compatibility	361
Changing the Names of Programs and Procedures	371
Creating C++ Classes for Use in ILE.	371
Mapping a C++ Class to a C Structure	372
Using C++ Objects in a C Program	373
Qualifying Library Calls.	376
Calling OPM Programs	376
Program Description	376
Program Structure.	376
Program Activation	377
Program Files	377
Invoking the ILE-OPM Program	382
Calling ILE Programs.	382
Program Description	382
Program Structure.	382

Program Activation	383
Program Files	384
Calling an ILE C++ Program	387
Calling an EPM C Program.	388
Calling ILE-Bindable APIs	389
Passing Operational Descriptors	392

Chapter 17. Using Teraspace. 395

Pointer Support in the C/C++ Compilers	395
Using Teraspace in Your C/C++ Programs	396

Part 7. Using International Locales and Coded Character Sets 401

Chapter 18. Internationalizing Your Program 403

Coded Character Set Identifier.	403
Source File Conversion	404
Creating a Source Physical File with a Coded Character Set Identifier	404
Changing the Coded Character Set Identifier (CCSID)	405
Converting String Literals in a Source File.	405
Using Unicode Support for Wide-Character Literals	406
Effect of Unicode on #pragma convert() Operations	407
Target CCSID Support	408
Literals, Comments, and Identifiers	408
Restrictions	409

Chapter 19. International Locale Support 411

Elements of a Language Environment	411
Locales	411
ILE C/C++ Support for Locales	412
ILE C/C++ Support for *CLD and *LOCALE Object Types.	412
C Locale Migration Table	412
POSIX Locale Definition and *LOCALE Support	415
LOCALETYPE Compiler Option	416
Creating Locales	416
Creating modules using LOCALETYPE(*LOCALE)	417
Categories Used in a Locale	417
Setting an Active Locale for your Application	417
Using Environment Variables to Set the Active Locale.	418
SAA and POSIX *Locale Definitions	419
Locale-Sensitive Run-Time Functions	419

Part 8. Appendixes 421

Appendix A. Improving Program Performance. 423

Data Types	423
Avoid Using the Volatile Qualifier	423
Replace Bit Fields with Other Data Types	424
Minimize the Use of Static and Global Variables	424
Use the Register Storage Class.	424

Classes	424
Performance Measurement	424
Use a Compiler Option to Enable Performance Measurement	424
Exception Handling	425
Reduce Exceptions	425
Turn Off C2M Messages during Record Input and Output	425
Use a Direct Monitor Handler.	426
Avoid Percolating Exceptions	426
Function Call Performance	426
Reduce the Number of Function Calls and Function Arguments	426
Input and Output Considerations	427
Use Record Input and Output Functions	427
ANSI C Record I/O	427
ILE C Record I/O	428
Use Input and Output Feedback Information	428
Block Records	429
Manipulate the System Buffer	429
Open Files Once for Both Input and Output	430
Reduce the Use of Shared Files	430
Reduce the Number of File Opens and Closes	430
Process Tape Files	430
Use Stream Input and Output Functions	431
Use C++ Input and Output Stream Classes	431
Use Physical Files Instead of Source Physical Files	431
Specify Library Names	432
Pointers	432
Open Pointers	432
Pointer Comparisons	432
Reduce Indirect Access	434
Shallow Copy and Deep Copy.	434
Space Considerations.	434
Choose Appropriate Data Types	434
Reduce Dynamic Memory Allocation Calls	435
Reduce Space Used for Padding	435
Remove Observability	437
Compress an Object	437
Activation Groups.	437
Calling Functions in Other Activation Groups	437
Reducing Program Start-Up Time	438
Program Control	438
Avoid Virtual Functions	438
Use Cheaper Operators	438
Compile-Time Performance Tips	438
Choosing Compiler Options	438
Run-Time Limits	439

Appendix B. Support for Data Description Specifications (DDS) . . . 441

Appendix C. Porting Programs to ILE C++ 443

Differences Between C and C++	443
Inter-language Calls	443
Binary Coded Decimal Class Library for OS/400	444
Header Files that Work with Both C and C++	451
Type Checking	453

Name Mangling	453
File Inclusion	453
Function Prototypes, Declarations and Pointers	453
Character Array Initialization	454
String Literals	454
Integrated File System	455
Set_Terminate is Scoped to an Activation Group	455

Appendix D. Using Templates in C++

Programs 457

Using Template Terms	457
How the Compiler Expands Templates	458
Generating Template Function Definitions	459
Including Defining Templates	460
Including Defining Templates Everywhere.	460
Structuring for Automatic Instantiation.	460
Manually Structuring for Single Instantiation	464

Appendix E. Casting with Run-Time

Type Information 465

Introducing RTTI	465
Using C++ Language Defined RTTI	466
The dynamic_cast Operator	466
Dynamic Casts with Pointers	466
Dynamic Casts with References	467
The typeid Operator	467
The type_info Class	469
Using RTTI in Constructors and Destructors	469
Understanding ILE C++ Extensions to RTTI	469
The extended_type_info Class	470

Bibliography 473

Notices 477

Programming Interface Information	478
Trademarks and Service Marks	478
Industry Standards	479

Index 481

About This Guide

This guide contains instructions on:

- Entering source statements
- Creating a program in two steps
- Creating a program in one step
- Running a program
- Debugging a program
- Managing streams and record files
- Writing programs that:
 - Use externally described files
 - Use database files and distributed data management files
 - Use device files
 - Handle exceptions
 - Call programs and procedures
 - Use pointers on the iSeries® system
- Internationalizing your program
- Using templates in C++ programs
- Porting programs to ILE C++
- Casting with run-time type information
- Using Teraspace support
- Customizing programs using locales

Who Should Use This Guide

This guide is for programmers who are familiar with the C and C++ programming languages and who plan to write or maintain ILE C/C++ applications. You need experience in using applicable iSeries menus and displays or Control Language (CL) commands.

Prerequisite and Related Information

Use the iSeries Information Center as your starting point for looking up iSeries and AS/400e technical information. You can access the Information Center in two ways:

- From the following Web site:
<http://www.ibm.com/eserver/iseries/infocenter>
- From CD-ROMs that ship with your Operating System/400 order:
iSeries Information Center, SK3T-4091-00. This package also includes the PDF versions of iSeries manuals, *iSeries Information Center: Supplemental Manuals*, SK3T-4092-00, which replaces the Softcopy Library CD-ROM.

The iSeries Information Center contains advisors and important topics such as CL commands, system application programming interfaces (APIs), logical partitions, clustering, Java™, TCP/IP, Web serving, and secured networks. It also includes links to related IBM® Redbooks and Internet links to other IBM Web sites such as the Technical Studio and the IBM home page.

Other information is listed in the “Bibliography” on page 473.

Install Licensed Program Information

On systems that make use of the ILE C/C++ compiler, the QSYSINC library must be installed.

A Note About Examples

Examples illustrating the use of the ILE C/C++ compilers are written in a simple style. The examples do not demonstrate all of the possible uses of C/C++ language constructs. Some examples are only code fragments and do not compile without additional code. All complete runnable examples begin with T1520. They can be found in the QCLE library, in source file QACSRC.

Most of the examples found in this guide are illustrated by entering Control Language (CL) commands on a CL command line. You can use a CL program to run most of the examples. See the member T1520INF in QCLE/QAINFO for information about running the examples in each chapter.

Control Language Commands

If you need prompting, type the CL command and press F4 (Prompt). If you need online help information, press F1 (Help) on the CL command prompt display. See *ILE C/C++ Compiler Reference* for command syntax for the CL commands including Integrated Language Environment CL commands. CL commands can be used in either batch or interactive mode, or from a CL program.

Note: You need object authority to use CL commands.

How to Send Your Comments

Your feedback is important in helping to provide the most accurate and high-quality information. IBM welcomes any comments about this book or any other iSeries documentation.

- If you prefer to send comments by mail, use the following address:

IBM Canada Ltd. Laboratory
Information Development
2G/KB7/1150/TOR
1150 Eglinton Avenue East
Toronto, Ontario, Canada M3C 1H7

If you are mailing a readers' comment form from a country other than the United States, you can give the form to the local IBM branch office or IBM representative for postage-paid mailing.

- If you prefer to send comments by FAX, use the following number:
 - 1-416-448-6161
- If you prefer to send comments electronically, use one of these e-mail addresses:
 - Comments on books:
torrcf@ca.ibm.com
IBMLink: to toribm(torrcf)
 - Comments on the iSeries 400 Information Center:
RCHINFOC@us.ibm.com

Be sure to include the following:

- The name of the book
- The publication number of the book
- The page number or topic to which your comment applies.

Figures

1. Program Creation in ILE	4
2. Class Hierarchy in the Example Program	8
3. Structure of the Example Program	9
4. ILE C Source to Add Integers and Print Characters	21
5. Structure of Program MYPROG	25
6. Example of a Basic Binder Listing	29
7. Calls between Program and Service Program	38
8. Display Module Information Screen for a Sample Module SEARCH	40
9. Binder Language Source File Generated for Module SEARCH	42
10. Binder Language Source File Generated by the RTVBNDSRC Command	44
11. Unresolved Import Requests in a Program With Circular References	45
12. Binder Language for Service Program SP1	46
13. Binder Language for Service Program SP2	47
14. Binder Language for Service Program SP1	49
15. Exports from Service Program COST	50
16. Import requests in Programs COSTDPT1 and COSTDPT2.	51
17. Basic Application Structure	55
18. Structure of the Application in ILE C	56
19. T1520DD1 — DDS Source for an Audit File	57
20. T1520CL1 — CL Source to Pass Variables to an ILE C Program	58
21. T1520CM1 — CL Command Source to Receive Input Data	58
22. ILE C Source to Call Functions in Other Modules	59
23. ILE C Source to Calculate Tax and Format Cost	60
24. ILE C Source to Write an Audit Trail	62
25. T1520IC4 — ILE C Source to Export Tax Rate Data	64
26. Binder Language Source to Export Tax Rate Data	64
27. Binder Language source to Export write_audit_trail Procedure	66
28. Sample Data for Program T1520PG1	67
29. Output File from Program T1520PG1	67
30. Calling Program XRUN2 Using the TFRCTL Command	72
31. Calling Program CALCOST from a User-Defined Command COST	73
32. T1520REP — ILE C Source to Pass Parameters to an ILE C Program	76
33. Running Programs in a Named Activation Group	79
34. Running Programs in Unnamed Activation Groups	80
35. Running a Service Program in the Activation Groups of Calling Programs	81
36. Module Source Display for DEBUGEX	108
37. Module Source Display After Stepping Into CPGM	108
38. Using EVAL to Change a Variable.	114
39. Sample EVAL Commands for Pointers, Variables, and Bit Fields	119
40. Sample EVAL Commands for C Structures, Unions and Enumerations	120
41. Sample EVAL Commands for System and Space Pointers	121
42. Sample EVAL Commands for C++ Expressions	122
43. Using EVAL with a Class Template	123
44. Using EVAL with a Function Template	123
45. Sample ILE Source Debugger and ILE C Application	131
46. System and Space Pointers	135
47. Debug Language Syntax (Program Flow and Program Data)	137
48. iSeries Data Management Records Mapping to an ILE C Stream File	142
49. ILE C Source to Open an ILE C Text Stream File	148
50. Writing to a Text Stream File	148
51. ILE C Source to Write Characters to a Text Stream File	149
52. Reading from a Text Stream File	149
53. ILE C Source to Read Characters from a Text Stream File	150
54. ILE C Source to Open a Binary Stream File	151
55. Writing to a Binary Stream File One Character at a Time	152
56. ILE C Source to Write Characters to a Binary Stream File	152
57. Reading from a Binary Stream File One Character at a Time	153
58. ILE C Source to Read Characters from a Binary Stream File	153
59. Updating a Binary Stream File with Data Longer than Record Length.	154
60. ILE C Source to Update a Binary Stream File with Data Longer than the Record Length	154
61. Updating a Binary Stream File With Data Shorter than Record Length.	155
62. ILE C Source to Update a Binary Stream File with Data Shorter than the Record Length	155
63. Writing to a Binary Stream File One Record at a Time	157
64. ILE C Source to Write to a Binary Stream File by Record.	157
65. Reading From a Binary Stream File a Record at a Time	158
66. ILE C Source to Read from a Binary Stream File by Record	158
67. The Integrated File System Interface	162
68. Comparison of a Stream File and a Record-Oriented File	168
69. iSeries Records Mapping to a C/C++ Stream File	168

70. Comparison of Text Stream and Binary Stream Contents	171	107. T1520DDG — DDS Source for a Subfile Display	233
71. Header Description	186	108. T1520SUB — ILE C Source to Use Subfiles	234
72. ILE C Source Using the Ivlchk Option	187	109. T1520DDA — DDS Source for Password and User ID	238
73. T1520DD3 — DDS Source for Program	188	110. T1520DDB — DDS Source to Send Password and User ID	238
74. Output Listing from the Program	188	111. T1520DDC — DDS Source to Receive Password and Userid	239
75. T1520DD8 — DDS Source for Customer Records	190	112. T1520ICF — ILE C Source to Send and Receive Data	239
76. T1520EDF — ILE C Source to Include an Externally Described Database File	191	113. T1520TGT — ILE C Source to Check Data is Sent and Returned	241
77. Output Listing from Program T1520EDF — Customer Master Record	192	114. T1520FCF — ILE C Source to Use First Character Forms Control	244
78. DDS Source for a Display File	192	115. Sample Source Statements for Program T1520TAP	247
79. Structure Definition for a Display File	193	116. T1520TAP — ILE C Source to Write to a Tape File	248
80. DDS Source for a Device File	193	117. T1520DSK — ILE C Source to Write Records to a Diskette File	251
81. Structure Definitions for a Device File	194	118. Error Handling for OPM and ILE	257
82. DDS Source for Indicators	195	119. Unhandled Exception Default Action	259
83. Structure Definition for Indicators	195	120. ILE C Source to Check for the Return Value of fopen()	260
84. Header Description Showing Comments for Indicators	196	121. ILE C Source to Check the errno Value for fopen()	260
85. Structure Definition for Multiple Formats	197	122. T1520DDJ — DDS Source for a Phone Book Display	262
86. Structure Definitions for a Device File	198	123. T1520EHD — ILE C Source to Handle Exceptions	264
87. Structure Definitions for BOTH Option	199	124. Exception Handler Priority	266
88. T1520ASP — ILE C Source to Process a Database Record File in Arrival Sequence	208	125. ILE C Source to Manage the State of a Signal Handler	267
89. T1520DD3 — DDS Source for Database Records	209	126. ILE C Source for Unhandled Exceptions	268
90. T1520KSP — ILE C Source to Process a Database Record File in Keyed Sequence	210	127. ILE C Source to Scope Direct Monitor Handlers	269
91. T1520DD4 — DDS Source for Database Records	211	128. ILE C Source to Use Exception Classes	270
92. T1520REC — ILE C Source to Process a Database File Using Record I/O Functions	211	129. ILE C Source to Handle Exceptions	271
93. T1520DD5 — DDS Source for Daily Transactions	213	130. T1520XH1 — ILE C Source to Use Direct Monitor Handlers — main()	272
94. T1520DD6 — DDS Source for Monthly Transactions	214	131. T1520XH2 — ILE C Source to Use Direct Monitor Handlers — Service Program	272
95. T1520DD7 — DDS Source for a Purchase Order Display	214	132. T1520XH3 — ILE C Source to Use Direct Monitors with Labels as Handlers	273
96. T1520COM — ILE C Source to Group File Operations Using Commitment Control	215	133. T1520ICA — ILE C Source to Use Direct Monitor Handlers	274
97. T1520DD9 — DDS Source for a Phone Book Display	223	134. ILE C Source to Nest Exceptions	278
98. T1520ID1 — ILE C Source to Specify Indicators as Part of the File Buffer	223	135. T1520XH4 — ILE C Source to Use Cancel Handlers	279
99. T1520DD0 — DDS Source for a Phone Book Display	225	136. T1520XH5 — ILE C Source to Use ILE Condition Handlers — main()	282
100. T1520ID2 — ILE C Source to Specify Indicators in a Separate Indicator Area	225	137. T1520XH6 — ILE C Source to Use ILE Condition Handlers — Service Program	282
101. T1520DDD — DDS Source for an I/O Display	226	138. T1520IC6 — ILE C Source to Use ILE Condition Handlers	283
102. T1520DEV — ILE C Source to Establish a Default Device	227	139. T1520IC7 — ILE C Source to Percolate a Message to Handle a Condition	284
103. T1520DDE — DDS Source for Name and Password Display	228	140. T1520IC8 — ILE C Source to Promote a Message to Handle a Condition	285
104. T1520CDV — ILE C Source to Change the Default Device	229	141. Resetting Signal Handlers	289
105. T1520DDF — DDS Source for a Feedback Display	231		
106. T1520FBK — ILE C Source to Use Feedback Information	232		

142.	Stacking Signal Handlers.	289	173.	ILE C Source to Suppress a Run-Time Exception	323
143.	T1520SIG — ILE C Source to Use Signal Handlers	290	174.	Basic Program Structure	328
144.	Example of Control Boundary When There is One Activation Group	293	175.	Structure of the Program in ILE C	329
145.	Example of Control Boundaries When You Have Multiple Activation Groups	294	176.	T1520DD2 — DDS Source for an Audit File	330
146.	Example of Control Boundaries in the Default Activation Group	296	177.	T1520CL2 — CL Source to Pass Variables to an ILE C Program	330
147.	T1520XH7 — ILE C Source for Exception Handling	297	178.	T1520CM2 — CL Command Source to Received Input Data	331
148.	ILE C Source to Declare Pointer Variables	301	179.	T1520IC5 — ILE C Source to Call COBOL AND RPG	331
149.	ILE C++ Source to Declare Pointer Variables	301	180.	T1520CB1 — OPM COBOL Source to Calculate Tax and Format Cost.	333
150.	ILE C Source to Declare a Pointer to a Bound Procedure.	302	181.	T1520RP1 — OPM RPG Source to Write the Audit Trail	335
151.	ILE C Source to Declare a Pointer to an iSeries 400 Program as a Function Pointer	302	182.	Basic Object Structure.	336
152.	ILE C++ Source to Declare a Pointer to an iSeries Program as a Function Pointer	303	183.	Integrated Language Environment Structure	337
153.	ILE C Source to Show iSeries Pointer Casting	304	184.	T1520CL3 — ILE CL Source to Pass Variables to an ILE C Program	338
154.	T1520DL8 — ILE C Source that Uses iSeries Pointers	305	185.	T1520ICB — ILE C Source to Call COBOL and RPG Procedures	339
155.	T1520DL9 — ILE C Source that Uses iSeries Pointers	306	186.	T1520CB2 — ILE COBOL Source to Calculate Tax and Format Cost	341
156.	ILE C Source to Convert Packed Decimals	310	187.	T1520RP2 — ILE RPG Source to Write the Audit Trail	342
157.	ILE C Source to Convert a Packed Decimal to a Packed Decimal with Smaller Precision	310	188.	SQUARE — CL Command Source to Receive Input Data	344
158.	ILE C Source to Convert a Packed Decimal to a Packed Decimal with Smaller Integral Part	310	189.	SQITF — ILE C Source to Pass an Argument by Value	344
159.	ILE C Source to Convert a Packed Decimal to a Packed Decimal with Smaller Integral Part and Smaller Precision	311	190.	SQ — ILE C Source to Perform Calculations and Return a Value	344
160.	ILE C Source to Convert a Packed Decimal with a Fractional Part to an Integer	311	191.	T1520API — ILE C Source to Call an ILE C Procedure.	346
161.	ILE C Source to Convert a Packed Decimal with Less than 10 Digits in the Integral Part to an Integer	311	192.	ILE C Source that Declares a Function that Requires Operational Descriptors	349
162.	ILE C Source to Convert a Packed Decimal with More than 10 Digits in the Integral Part to an Integer	312	193.	ILE C Source to Generate Operational Descriptors	350
163.	ILE C Source to Convert a Packed Decimal with More than 10 Digits in Both Parts to an Integer.	312	194.	ILE C Source to Call a Function with Operational Descriptors	350
164.	ILE C Source to Convert a Packed Decimal to a Floating Point.	312	195.	ILE C Source to Determine the Strong Arguments in a Function.	350
165.	ILE C Source to Pass Packed Decimal Variable to a Function.	313	196.	Program and Procedure Calls on the Call Stack	353
166.	ILE C Source to Pass a Pointer to a Packed Decimal Value to a Function	314	197.	ILE C++ Procedures Cannot Call Other Active ILE COBOL Procedures	353
167.	ILE C Source for an ILE C Program that Passes Packed Decimal Data	315	198.	Basic Program Structure	376
168.	COBOL Source that Receives Packed Decimal Data from an ILE C Program	316	199.	Structure of the Program in ILE C++	377
169.	ILE C Source to Use the va_arg Macro with a Packed Decimal Data Type	317	200.	ILE Structure.	383
170.	ILE C Source to Write Packed Decimal Constants to a File and Scan Them Back	318	201.	Basic Object Structure.	383
171.	ILE C Source to Print Packed Decimal Constants	320	202.	Source File CCSID Conversion.	404
172.	Packed Decimal Warnings and Error Conditions	321	203.	T1520CCS — ILE C Source to Convert Strings and Literals	406
			204.	Using ANSI C Record I/O	428
			205.	Using ILE C Record I/O.	428
			206.	I/O Feedback Information	429
			207.	Using the System Buffer	429
			208.	Opening a File Twice	430
			209.	Opening a File Once	430
			210.	Using printf()	431
			211.	Using printf() to Reduce Function Calls	431

Tables

1. Programming Languages Supported by the iSeries family	3	20. ILE C++ Data-Type Compatibility with ILE RPG	361
2. Trigraphs	22	21. ILE C++ Data-Type Compatibility with ILE COBOL	362
3. Parameters for CRTPGM Command and their Default Values	27	22. ILE C++ Data-Type Compatibility with ILE CL	364
4. Sections of the Binder Listing based on the DETAIL Parameter	28	23. ILE C++ Data-Type Compatibility with OPM RPG	365
5. Parameters and Default Values for CRTSRVPGM Command	34	24. ILE C++ Data-Type Compatibility with OPM COBOL	366
6. Integrated File System Compiles	173	25. ILE C++ Data-Type Compatibility with CL	367
7. Data Management File System Compiles	173	26. Arguments Passed From a Command Line CL Call to an ILE C++ Program	368
8. INCDIR Command Parameter	175	27. CL Constants Passed from a Compiled CL Program to an ILE C++ Program	368
9. INCLUDE Environment Variable	176	28. CL Variables Passed from a Compiled CL Program to an ILE C++ Program	369
10. Include Search Order	176	29. C Locale Migration Table	412
11. Parameter Values	177	30. Categories Used in a Locale.	417
12. INCDIRFIRST Command Options.	178	31. Locale-Sensitive Run-Time Functions	419
13. Lock States for Open Modes	204	32. Compiler Options for Performance	439
14. Handling Overflow From a Packed Decimal to a Smaller Target.	313	33. Flag Meanings for Printing the Value of a _DecimalT Template Class Object	448
15. Program Calling Conventions	326	34. Comparing Packed Structures	452
16. Argument Passing for Integrated Language Environment Procedures	346	35. typeid operations	468
17. Effects of Various Linkage Specifications	355		
18. Default Argument Passing Style for ILE Programs	360		
19. Default Argument Passing Style for ILE Procedures	360		

Part 1. Introduction

This part introduces:

- Integrated Language Environment (ILE) and OS/400 Operating System Programming Features
- An overview of program creation in ILE
- Main Features of the C++ Language
- Using C++ for Object-Oriented Programming

Chapter 1. Introduction to the ILE C/C++ Compiler

The ILE C/C++ Compiler supports program development on iSeries systems in both C and C++ programming languages.

C and C++ are two of the programming languages supported by the *Integrated Language Environment (ILE)*. C++ provides additional features to those found in the C language. These features include additional keywords, parameterized types (templates), support of object oriented programming via classes, and stricter type checking.

ILE and OS/400 Operating System Programming Features

C and C++ are two of the programming languages supported in the *Integrated Language Environment (ILE)*. ILE, together with the OS/400 operating system, provides a wide range of support for serious program development.

ILE provides you with advantages in these areas of program development:

- Support for multiple programming languages
- Program creation
- Program and resource management
- Program calls
- Program debugging
- Bindable APIs

Programming Languages Supported by the OS/400 Operating System

ILE is an approach to programming on the iSeries systems. You can build mixed-language programs that are composed of modules written in any ILE programming language. The ILE family of compilers includes: ILE C++, ILE C, ILE RPG, ILE COBOL, and ILE CL. Table 1 lists the programming languages supported by the OS/400 operating system.

Table 1. Programming Languages Supported by the iSeries family

Integrated Language Environment (ILE)	Original Program Model (OPM)	Extended Program Model (EPM)
C++	BASIC (PRPQ)	C
C	CL	FORTRAN
CL	COBOL	PASCAL (PRPQ)
COBOL	PL/I (PRPQ)	
RPG	RPG	

Program Creation

ILE program creation consists of:

1. Compiling source code into modules
2. Binding (combining) one or more modules into a program object.

You can create and maintain mixed-language programs because you can combine modules from any ILE language.

You can create a *binding directory* to contain the names of modules and service programs that your ILE C++ program or service program may need. A binding directory can reduce program size because modules or service programs listed in a binding directory are used only if needed.

You can bind modules into service programs (*SRVPGM). *Service programs* are a means of packaging callable routines (functions or procedures) into a separately bound program. The use of service programs provides modularity and improves maintainability. You can use off-the-shelf modules developed by third parties or package your own modules for third-party use. Service programs are created by a compiler option that includes the Create Service Program (CRTSRVPGM) command.

Figure 1 shows the process of creating an ILE program through compiler and binder invocation.

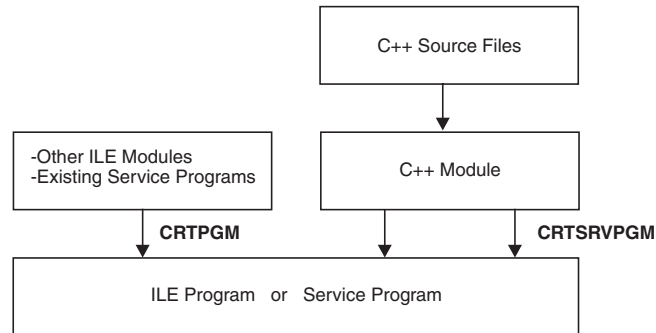


Figure 1. Program Creation in ILE

Once a program is created, update it using the Update Program (UPDPGM) or Update Service Program (UPDSRVPGM) commands. These commands are useful because you only need to have the new or changed modules available when you want to update the program.

Program and Resource Management

ILE provides a common basis for:

- Managing program flow
- Sharing resources
- Using Application Program Interfaces (APIs)
- Handling exceptions during a program's run time

ILE programs and service programs are activated into *activation groups* you specify at the time you create a program. The process of getting a program or service program ready to run is known as activation. Activation allocates resources within a job so that one or more programs can run in that space. When a program is called, the system activates it into an activation group. If the specified activation group for a program does not exist when the program is called, it is created within the job to hold the program's activation.

An activation group is the key element in governing an ILE program's resources and behavior. You can scope commitment-control operations to the activation

group level. You can scope file overrides and shared open data paths to the activation group of the running program. The behavior of a program upon termination is affected by the activation group in which the program runs.

Program Calls

In ILE, you can write programs in which ILE C++ programs, OPM and EPM programs interrelate through the use of *dynamic program calls*. When using such calls, the calling program specifies the name of the called program. This name is resolved to an address at run time, just before the calling program passes control to the called program.

You can write programs which interrelate through faster *static procedure calls*. A procedure is a self-contained set of code that performs a task and then returns to the caller. An ILE C++ module consists of one or more procedures. Because the procedure names are resolved at bind time (that is, when you create the program), static calls are faster than dynamic calls.

Static calls allow operational descriptors. Operational descriptors are used to call bindable APIs or procedures written in other ILE languages.

See Chapter 16, “Calling Conventions” on page 325 for information on calls between programs and procedures.

Program Debugging

In ILE, you can perform source-level debugging on any program written in one or more ILE languages, provided program was compiled with debug information. You can control the flow of a program by using debug commands while the program is running. You can set conditional and unconditional breakpoints prior to running the program. After calling the program, you can step through a specified number of statements and display or change variables. When a program stops because of a breakpoint, a step command, or a run-time error, the pertinent module is displayed at the point where the program stopped. At that point, you can enter more debug commands.

Bindable APIs

ILE offers a number of bindable APIs that supplement ILE C/C++ functions. Bindable APIs provide program calling and activation capability, condition and storage management, math functions, and dynamic screen management. The *System API Reference* contains information on bindable APIs.

Main Features of the C++ Language

The C++ programming language is based on the C language.

Although C++ is a descendant of the C language, the two languages are not always compatible. Please refer to the ILE C/C++ Language Reference for more information.

In C++, you can develop new data types that contain functional descriptions (member functions) as well as data representations. These new data types are called *classes*. The work of developing such classes is known as *data abstraction*. You can work with a combination of classes from established class libraries, develop your own classes, or derive new classes from existing classes by adding data descriptions and functions. New classes can contain (*inherit*) properties from one or

more classes. The classes describe the data types and functions available, but they can hide (*encapsulate*) the implementation details from the client programs.

You can define a series of functions with different argument types that all use the same function name. This is called *function overloading*. A function can have the same name and argument types in base and derived classes.

Declaring a class member function in a base class allows you to override its implementation in a derived class. If you use virtual functions, class-dependent behavior may be determined at run time. This ability to select functions at run time, depending on data types, is called *polymorphism*.

You can redefine the meaning of the basic language operators so that they can perform operations on user-defined classes (new data types), in addition to operations on system-defined data types, such as `int`, `char`, and `float`. Adding properties to operators for new data types is called *operator overloading*.

The C++ language provides templates and several keywords not found in the C language. Other features include *try-catch-throw* exception handling, stricter type checking and more versatile access to data and functions compared to the C language.

Using C++ for Object-Oriented Programming

The features that provide the C++ language with the facilities for object-oriented programming are the ability to:

- Define new data types, including those that contain both data elements and a set of operations applicable to them (data abstraction).
- Inherit data elements and operations from other data types (inheritance).
- Hide the data-type implementation details so that client programs use a well-defined interface (encapsulation).
- Select functions at run time depending on data types (polymorphism).

Coding a C++ Program

This section describes an example program that shows you the main features and structure of the C++ language. The source code shows these C++ programming techniques:

- Defining an abstract base class.
- Deriving classes from a base class.
- Deriving classes from multiple base classes (multiple inheritance).
- Defining private and public members, constructors, and functions of a class.
- Declaring function prototypes.
- Declaring inline functions in a class and in the program.
- Overloading functions.
- Using virtual functions.
- Using the I/O stream library for program output.

Program Description

The program produces a company's monthly pay summary. It processes data on four different types of employees:

- Managers on a yearly salary.
- Sales managers who get a yearly salary plus a commission for units sold.

- Regular employees who are paid an hourly wage.
- Sales persons who are paid a commission based on the number of units they sell.

The program displays the basic monthly pay for each class of employee, the name and identification number for each employee, and the amount each employee was paid for the month. It calculates the total amount paid in salary, wages, and commission for the month.

Program Structure

The program files are:

- A user-defined header file `compclas.h` containing the class declarations.
- A C++ source file `compfunc.cpp` containing class member function definitions.
- A C++ source file `comprec.cpp` containing the main program logic.
- A system header file `iostream.h` from the Standard Class library containing standard input and output functions.

User-Defined Header File: In the header file `compclas.h`, five classes are declared:

- An abstract base class `employee` containing the employee's name and identification number.
- Three derived classes `manager`, `regular_emp`, and `sales_person` contain constructor functions and functions for printing out pay information.
- A class `sales_mgr` that uses the pay functions from both the `sales_person` and `manager` classes in its `pay` function to demonstrate multiple inheritance.

Each class contains private data on salaries, or wages and hours worked, or commission and units sold, and on public functions that use the data. The class structure demonstrates the private and public aspects of a class. Each class contains one inlined function to demonstrate inlining within a class. Figure 2 on page 8 shows the class hierarchy.

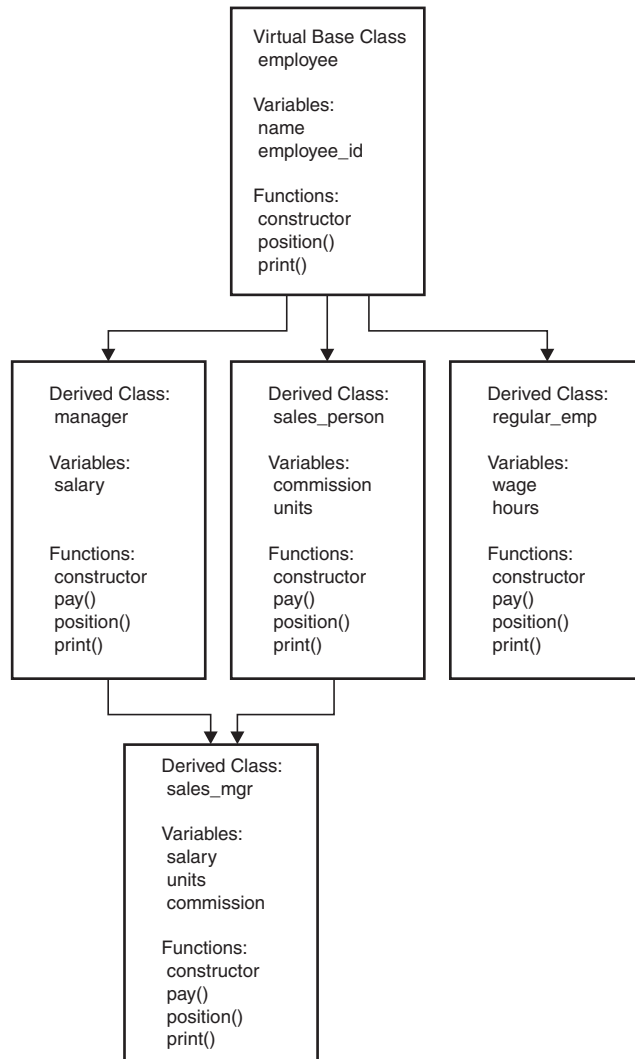


Figure 2. Class Hierarchy in the Example Program

C++ Source Files: The file `compfunc.cpp` contains the definitions for the member functions of the classes used in this program.

The file `comprec.cpp` contains the logic for this program. It defines a function `payout()` that prints out a basic salary for each category of employee.

The function `payout()` demonstrates how you prototype and overload a function. Each class of employee has a different number of parameters, but the same function name is used for all employee classes. The functions are prototyped before the definition of the `main()` function. They are defined in the body of `comprec.cpp` after the definition of the `main()` function.

The `main()` function in `comprec.cpp` defines an instance of each of the four classes of employee: manager, sales manager, regular employee, and sales person, and uses the functions defined for each of these classes to display information about four typical employees. Figure 3 on page 9 shows the program structure.

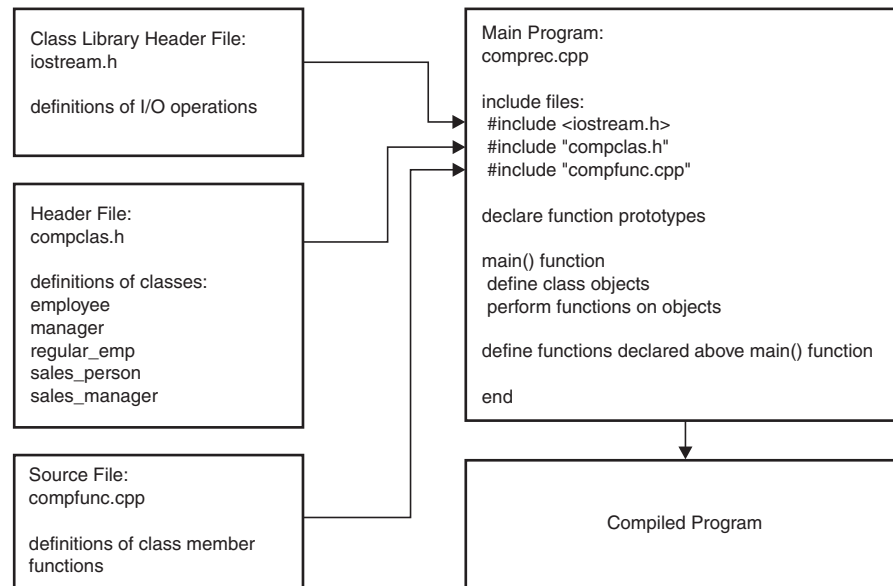


Figure 3. Structure of the Example Program

Program Files

The following sections show the source code for each of the files that compose this program.

Header File: The header file `compclas.h` contains definitions of classes that are used in the main program `comprec.cpp`. See “Main Program Source File” on page 13.

```

// compclas.h -- class definitions for comprec.cpp ***

#ifndef _COMPCLAS_H
# define _COMPCLAS_H 1

// Abstract base class definition
class employee {
private:
    char * n;
    int x;
    employee( const employee & lhs);
    employee& operator =( const employee & lhs);

protected:
    const char * name() { return n; }
    int employee_id () { return x; }

public:
    // Constructors for class employee.
    employee() : n(""), x(0) {};
    employee(char * n, int id);

    virtual ~employee ();
    virtual double pay() =0;
    virtual void dump() =0;
};
// End of abstract base class definition

// Derived class definitions.

// Derive a class manager from class employee
  
```

```

class manager : virtual public employee {
protected:
    double salary;
    manager(double sal);

public:
    // Constructors for class manager.
    manager(char *n, int id, double sal);

    // Member functions for class manager.
    double pay();
    friend ostream& operator << (ostream& out, manager & lhs);
    void dump() { cout << *this; }
};

// End of manager class definition.

// Derive a class regular_emp from class employee
class regular_emp : virtual public employee {
private:
    double wage, hours;

public:
    // Constructor
    regular_emp(char *n, int id, double wg, double hrs) ;
    // Member functions
    double pay();
    friend ostream& operator << (ostream& out, regular_emp & lhs);
    void dump() { cout << *this; }
};

// End of regular_emp class definition.

// Derive a class sales_person from class employee
class sales_person : virtual public employee {
protected:
    double commission;
    int units;
    sales_person(double com, double nts);

public:
    // Constructors
    sales_person(char *n, int id,
                 double com, double nts);
    // Member functions
    double pay();
    friend ostream& operator << (ostream& out, sales_person & lhs);
    void dump() { cout << *this; }
};

// End of sales_person class definition.

// Derive a class sales_mgr from the manager and sales_person classes
class sales_mgr : public manager, public sales_person {
public:
    // Constructors
    sales_mgr(char *n, int id, double sal, double comm, double nts);
    // Member functions
    double pay();
    friend ostream& operator << (ostream& out, sales_mgr & lhs) ;
    void dump() { cout << *this; }
};

// End of sales_mgr class definition.

#endif

```

Program References:

1 An abstract base class must have at least one pure virtual function. You cannot declare instances of an abstract base class.

2 Classes can have public, private, and protected data members and functions. Data can be private (hidden) while the functions that make use of this data are public. Hiding data is a way of making the data available through a limited set of functions, while at the same time protecting the data from change by an unauthorized user.

3 Constructors have the same name as the class. For virtual base classes with multiple inheritance, you must have one constructor that takes no arguments. You can have as many constructors as you need, but each must have different numbers or types of arguments.

4 Destructors have the same name as the class, preceded by a tilde (~) character. The destructor is ~employee. You cannot specify arguments or a return type for a destructor. The destructor destroys members of the class immediately before the class object is destroyed.

5 Virtual functions defined in a base class of a hierarchy are often not intended to be invoked. They are *pure virtual functions*; functions declared with the keyword *virtual*, which define a return type, and which end with = 0. Each derived class must override the definition of each pure virtual function of its base classes. Each derived class must define a function with the same name and arguments. These functions must return the same type as the corresponding function in the base class.

A class that contains at least one pure virtual function can be used only as a base class for subsequent derivations of other classes.

6 Derived classes inherit the data members and functions from the base class. The colon operator (:), followed by the name of the base class, indicates that the class being defined is a derived class.

7 Functions defined within the class definition are automatically inlined. You do not need the keyword **inline**.

8 A class with multiple inheritance inherits data structures and member functions from more than one class. Class sales_mgr has the properties of both a manager class and a sales_person class. Class sales_mgr represents someone who earns both a regular salary (such as a manager) and earns a commission (such as a sales_person).

C++ Source File: The file compfunc.cpp contains the function definitions for the member functions of the classes that are used in the main program comprec.cpp. See “Main Program Source File” on page 13.

```
//compfunc.cpp -- definitions for class member functions
```

```
#include <string.h>
#include <iostream.h>
#include "compclas.h"
```

```
// Constructor definition for class manager.
employee::employee(char * name, int id)
: n (new char [strlen (name) + 1 ]), x(id) {
    strcpy (n, name);
};
employee::~~employee() { delete [] n; }
```

```
// Note different way to initialize.
manager::manager(char *n, int id, double sal)
: employee(n, id), salary(sal){}
```

9
10

```

manager::manager(double sal)
    : salary(sal){}

// Member function definitions for class manager
double manager::pay() {
    return salary/12 ;
}

ostream& operator << (ostream& out, manager & lhs) {
    out << endl << lhs.name()
        << " is a manager with employee number " << lhs.employee_id () << endl

        << "makes " << lhs.salary << " per year." << endl

        << lhs.manager::name() << " made " << lhs.pay() << " dollars this month."
        << endl << endl;
    return out;
}

// Constructor for class regular_emp.
regular_emp::regular_emp(char *n, int id, double wg, double hrs)
    : employee(n, id), wage(wg), hours(hrs) {}

// Member function definitions for class reg_emp
double regular_emp::pay() {
    return wage*hours;
}

ostream& operator << (ostream& out, regular_emp & lhs) {
    out << endl << lhs.name() << " is a regular employee with employee number "
        << lhs.employee_id() << endl

        << "makes " << lhs.wage << " dollars per hour" << endl

        << "and worked " << lhs.hours << " hours this month. " << endl

        << lhs.name() <<" made " << lhs.pay()
        << " dollars this month." << endl << endl;
    return out;
}

// Constructor definition for class sales_person.
sales_person::sales_person(char *n, int id,
    double comm, double nts)
    : employee(n, id), commission(comm), units(nts){}

sales_person::sales_person(double comm, double nts)
    : commission(comm), units(nts){}

// Member function definitions for class sales_person.
double sales_person::pay() {
    return commission*units;
}

ostream& operator << (ostream& out, sales_person & lhs) {
    out << endl << lhs.name() << " is a sales person with employee number "
        << lhs.employee_id() << endl

        << "works for a straight commission of "
        << lhs.commission << endl

        << "dollars per unit sold and sold " << lhs.units
        << " units this month." << endl

        << lhs.name() <<" made " << lhs.pay()
        << " dollars this month." << endl << endl;
    return out;
}

```



```

// Constructor for class sales_mgr 11
sales_mgr::sales_mgr(char *n, int id,
                    double sal, double comm, double nts)
    :employee(n, id),
    manager(sal),
    sales_person(comm, nts) { }

// Member function definitions for class sales_mgr. 12
double sales_mgr::pay(){
    return manager::pay() + sales_person::pay()
;}

ostream& operator << (ostream& out, sales_mgr & lhs) {
    out << endl << lhs.name() << " is a sales manager with employee number "
    << lhs.employee_id() << endl

    << "makes " << lhs.salary << " per year and earns a commission of "
    << lhs.commission << " dollars per unit sold." << endl

    << lhs.name() << " was responsible for sales of "
    << lhs.units << " units this month." << endl

    << lhs.name() << " made " << lhs.pay()
    << " dollars this month." << endl << endl;
    return out;
}

```

Program References:

- 9** The scope resolution operator (::) in this function definition indicates that the constructor function employee for class employee is defined.
- 10** This constructor initializes the data types declared in the class definition.
- 11** For classes with multiple inheritance, the constructor must fully initialize the class without creating any ambiguity.
- 12** Because sales_mgr is a derived class with multiple inheritance, there can be ambiguity about which values to use. To avoid ambiguity, explicitly qualify the function by using the scope resolution operator (::).

Main Program Source File: The C++ source file comprec.cpp contains the logic of this program.

```

//comprec.cpp 13
#include <iostream.h>
#include <strstream.h>
#include "compclas.h"
#include "compfunc.cpp" // for the inline functions

static void payout(double); 14
static void payout(double, double); 15
static void payout(double, double, double); 16
inline void title() { 16
    cout << "Monthly Employee Pay Report" << endl << endl;
};
// Start of main function 17
const char * pwcharset = "abcdefghijklmnopqrstuvwxyz"
                        "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
                        "0123456789\r\n";

int main() { 18
    const double managers_pay = 1510.35;
    const double reg_emp_pay = 25.75;
    const double reg_emp_hrs = 40.00;
    const double sales_mgr_units = 200;
    const double monthly_salary = 800.00;
}

```

```

const double commission = 1.00;
const double units = 150;
cout.setf(ios::fixed);
cout.precision(2);
title();

// Optionally, build in a security check
char val [512] ;
cout << "To view data on employees, "
    << "type your password (mona) and press Enter:" << endl;
cin >> val ;
char * p = val;
char * where;
int check = 32679;
while (*p) {
    where = strchr(pwcharset,*p);
    if (where) check -= (where-pwcharset+7);
    ++p;
}
check &= 32767;
if (check != 9196) {
    cout << "Wrong password, bye." << endl;
// cout << "check is " << check << endl;
    return 999;
}

payout(managers_pay);
payout(reg_emp_pay, reg_emp_hrs);
payout(monthly_salary, commission, units);

// Define instances of classes and call their member functions
manager smith("Jack Smith", 123, 28020);
cout << smith << endl;

regular_emp james("Everett James", 456, 12,160) ;
cout << james << endl;

sales_person doe("Jackson Doe", 101, 31,65);
cout << doe << endl;

sales_mgr stevens("Jennifer Stevens", 789, 28000, 4, 105) ;
cout << stevens << endl;
double sal1, sal2, sal3, sal4;
sal1 = smith.pay();
sal2 = james.pay();
sal3 = doe.pay();
sal4 = stevens.pay();

// Use the values returned from the pay functions.
cout << "Total wages paid to these employees was: "
    << (sal1+sal2+sal3+sal4)
    << " dollars" << endl << endl ;

// Alternate method of defining instances of a class
employee * ep[] = {
    new manager    ("Jack Smith", 123, 28020),
    new regular_emp ("Everett James", 456, 12,160),
    new sales_person ("Jackson Doe", 101, 31,65),
    new sales_mgr   ("Jennifer Stevens", 789, 28000, 4, 105),
    0
} ;

double sal=0;
for (int i=0; ep[i]; ++i) {
    ep[i]->dump();
    sal+=ep[i]->pay();
    delete ep[i];
}

```

```

}

cout << "Total wages paid to these employees was: "
    << sal
    << " dollars" << endl << endl;
return 0;
}
// End of function main

// Definition of payout functions
// An overloaded function with one, two, and 3 arguments
void payout(double managers_pay) {
    cout << "The basic salary for a manager is: "
        << managers_pay << " dollars per month." << endl << endl;
}
void payout(double reg_emp_pay, double reg_emp_hrs) {
    double reg_monthly_pay;
    reg_monthly_pay = reg_emp_pay * reg_emp_hrs;
    cout << "The basic pay for a regular employee is: "
        << reg_monthly_pay << " dollars per month." << endl << endl;
}
void payout(double monthly_salary, double commission, double units) {
    double reg_monthly_pay;
    reg_monthly_pay = (monthly_salary + (commission*units));
    cout << "The basic pay for a sales manager is: "
        << reg_monthly_pay << " dollars per month." << endl << endl;
}

```

Program References:

13 The first two preprocessor directives let this program make use of system include files that declare standard library functions such as the `cout` and `cin` objects. The compiler searches for the files named in the `#include` directives.

The third and fourth preprocessor directives name the user-defined include file `compclas.h` and the source file `comfunc.cpp` that define the classes and member functions used by `comprec.cpp`.

14 All functions must be declared before they can be used. This statement declares a function `payout` that takes an argument of type `double` and does not return a value.

The function has been declared `static` to limit its scope to this file only. One reason to use the static storage class for functions is to avoid possible name conflicts.

15 An overloaded function has the same name as another function but different types or numbers of arguments. Each function uses the same type of argument (`double`), but each has a different number of arguments.

16 The keyword **`inline`** specifies a function whose body is to be expanded at each point of call to the function, so that the call is replaced by the function body itself. Within a class definition, the function is automatically inlined if you include its definition in the class definition. Specifying **`inline`** may not inline the function. The compiler decides whether the function is inlined.

You can control inlining with the `/Oi+` compiler option.

17 Each program must have a `main` function. Within the braces are statements that make up the main body of the program. `main` takes no arguments and returns an `int`.

18 All variables must be declared before they are used.

Declaring a variable as `const` ensures that the program cannot inadvertently change the value of this variable.

In your program, the values provided here would probably be stored in a database file, since they may be revised more often than the code itself.

19 The `cout` object is part of the I/O stream class library. Its member functions are used to set decimal places for the output.

20 You can code a security feature such as a password.

21 An instance of the class provides a name for the instance and the values required by one of the constructors of the class. An instance `smith` of class `manager` is initialized with the three parameters: `name`, `employee_id`, and `salary`.

22 These variables are declared in the body of the program, because this is where they are used.

23 This is one way to use the return values from the pay functions from each class. To streamline the program, you can include the functions in the next step without having to define the four variables `sal1` to `sal4`.

24 Expressions and function calls can be used as input to the `cout` object.

25 As an alternative to creating each instance of a class separately, you can use an array instead.

Instead of calling member functions separately for each member of the array, you can use a `for` loop.

Functions must be defined within a class or at file scope. They must be declared (prototyped) before they are used.

Program Output: The output of the sample program is:

Monthly Employee Pay Report

To view data on employees, type your password and press Enter:
The basic salary for a manager is: 1510.35 dollars per month.

The basic pay for a regular employee is: 1030.00 dollars per month.

The basic pay for a sales manager is: 950.00 dollars per month.

Jack Smith is a manager with employee number 123
makes 28020.00 per year.
Jack Smith made 2335.00 dollars this month.

Everett James is a regular employee with employee number 456
makes 12.00 dollars per hour
and worked 160.00 hours this month.
Everett James made 1920.00 dollars this month.

Jackson Doe is a sales person with employee number 101
works for a straight commission of 31.00
dollars per unit sold and sold 65 units this month.
Jackson Doe made 2015.00 dollars this month.

Jennifer Stevens is a sales manager with employee number 789
makes 28000.00 per year and earns a commission of 4.00 dollars per unit sold.
Jennifer Stevens was responsible for sales of 105 units this month.
Jennifer Stevens made 2753.33 dollars this month.

Total wages paid to these employees was: 9023.33 dollars

Part 2. Creating and Compiling Programs

This part introduces:

- How to create programs with ILE C/C++ Compilers
- How to use the compiler and binder programs to create program modules and executables for your own applications
- How to create ILE service programs
- How to run ILE programs

Chapter 2. Creating a Program

This chapter describes how to:

- Enter source statements:
 - Create a library and source physical file
 - Enter source statements into a member of a source physical file
- Create a program in one step
- Create a program in two steps:
- Create a program from source statements that are stored in an Integrated File System file.

Introducing the Program Development Process

During the development process, an iSeries program passes through five stages:

1. Preparing
2. Compiling
3. Binding
4. Running
5. Debugging

This process is not continuous. You can compile, correct compile-time errors, modify, and recompile the program several times before binding it.

Preparing a Program

Preparing a program involves designing, writing, and creating source code. See “Entering Source Statements” on page 20 for more information about creating source code.

Compiling

Issue the compile command against your source, and fix any compile errors that arise. You can see the errors either as messages in the job log or in the listing (if you chose to create one).

The ILE C/C++ compiler includes the following compile commands:

Compile Command	Use	Description
CRTCMOD	Create C Module	The Create Module command creates a module object. If your program will include objects from more than one source file, you must use the Create Module command for each source file, and then run CRTPGM specifying all the required *MODULEs to create the bound program.
CRTCPPMOD	Create C++ Module	
CRTBNDC	Create Bound C Program	The Create Bound Program command performs both the module creation and the binding operation in one step, and produces a *PGM object from a single source file.
CRTBNDCPP	Create Bound C++ Program	
Note: The compile command might also originate in a CL program or makefile.		

See *ILE C/C++ Compiler Reference* for more information about the Create Module and Create Bound Program commands and their options.

Note: In the following pages:

1. CRTCMOD and/or CRTCPPMOD may be referred to as simply the "Create Module" command.
2. CRTBNDC and/or CRTBNDCPP may be referred to as simply the "Create Bound Program" command.
3. Examples may show the use of either of the C or C++ versions of the Create Module and Create Bound Program commands. Unless specifically stated otherwise, both C and C++ versions of these commands function in the same way and can be used interchangeably, according to the language of the source program being compiled.

Binding

If you created modules during compilation, you need to bind the module objects together using the Create Program (CRTPGM) or Create Service Program (CRTSRVPGM) commands. The result is an executable *PGM or *SRVPGM object.

Binding combines one or more modules into a program (*PGM) or a service program (*SRVPGM). Modules written in ILE C or C++ can be bound to modules written in any other ILE language. C++ programs can use routines from C++ class libraries, C libraries, and any ILE service program. The binder resolves addresses within each module, import requests and export offers between modules that are being bound together.

Once a program is created, you can later update it using the Update Program (UPDPM) or Update Service Program (UPDSRVPM) commands. These commands are useful, because you only need to have the new or changed modules available when you want to update the program.

Running

*CMD objects are run, while *PGM objects are called. For example, to run HELLO *CMD, type HELLO on the QCMD command line and press ENTER. To run HELLO *PGM, type CALL HELLO on the QCMD line and press ENTER.

Entering Source Statements

Before you can start an edit session and enter your source statements, you must create a library and a source physical file. You can also compile source statements from Integrated File System (IFS) files. We strongly recommend that source code be kept in IFS stream files. See "Using the Integrated File System" on page 171 for details.

You can use the Start Programming Development Manager (STRPDM) command to start an edit session, and enter your source statements.

Besides PDM, there are several other ways to enter your source:

- The Copy File (CPYF) command.
- The Start Source Entry Utility (STRSEU) command.
- The Programmer Menu.

This is by no means an exhaustive list. There are other ways of creating source and placing it on an iSeries system, including NFS, and ftp.

Example

The following example shows you how to create a library, a source physical file, a member, start an edit session, enter source statements, and save the member.

1. To create a library called MYLIB, type:

```
CRTLIB LIB(MYLIB)
```

2. To create a source physical file called QCSRC in library MYLIB, type

```
CRTSRCPF FILE(MYLIB/QCSRC) TEXT('Source physical file for all ILE C programs')
```

QCSRC is the default source file name for ILE C commands that are used to create modules and programs. For ILE C++ commands, the corresponding default is QCPPSRC. For information about how to copy this file to an Integrated File System file, see “Using the Integrated File System” on page 171.

3. To start an edit session type:

```
STRPDM
```

4. Choose option 3 (Work with members); specify the source file name QCSRC, and the library MYLIB.

5. Press F6 (Create), enter the member name T1520ALP, and source type C. The SEU Edit display appears ready for you to enter your source statements.

6. Type the following source into your SEU Edit display. Trigraphs can be used in place of square brackets, as demonstrated in this example.

```
/* This program reads input from the terminal, displays characters, */
/* and sums and prints the digits. Enter a "+" character to      */
/* indicate EOF (end-of-file).                                   */

#define MAXLEN 60          /* The maximum input line size.    */

#include <stdio.h>
#include <ctype.h>

void main(void)
{
    int c;
    int i = 0, j = 0;
    int sum = 0;
    int count, cnt;
    int num[MAXLEN];        /* An array of digits.      */
    char letter??(MAXLEN??); /* An array of characters. */

    while ( ( c = getchar( ) ) != '+' )
    {
        if ( isalpha ( c ) ) /* A test for an alphabetic */
        { /* character.      */
            letter[i++] = c;
        }
        else if ( isdigit ( c ) ) /* A test for a decimal digit. */
        {
            num??(j+??) = c - '0';
        }
    }
}
```

Figure 4. ILE C Source to Add Integers and Print Characters (Part 1 of 2)

```

printf ( "Characters are " );
for ( count = 0; count < i; ++count )
{
    printf ( "%c", letter[count] );
}
printf( "\nSum of Digits is " );
for ( cnt = 0; cnt < j; ++cnt )
{
    sum += num[cnt];
}
printf ( "%d\n", sum );
}

```

Figure 4. ILE C Source to Add Integers and Print Characters (Part 2 of 2)

7. Press F3 (Exit) to go to the Exit display. Type Y (Yes) to save the member T1520ALP.

The ILE C compiler recognizes source code written in any single-byte EBCDIC CCSID (Coded Character Set Identifier) except CCSID 290, 905 and 1026. See Chapter 18, "Internationalizing Your Program" on page 403 for information on CCSIDs. You can use the trigraphs shown in Table 2 in place of characters in the C character set that are not available on your keyboard.

Table 2. Trigraphs

C Character	Trigraph
#	??=
[??(
]	??)
{	??<
}	??>
\	??/
	??!
^	??'
~	??-

The C compiler also supports digraphs. The C++ compiler does not support digraphs.

Creating a Program in One Step

You can use the CRTBNDC and CRTBNDCPP Create Bound Program commands to create a program (*PGM object) in one step.

The Create Bound Program commands combine the steps of compiling and binding. Using them is the same as first calling the CRTCMOD or CRTCPPMOD Create Module command, then calling the Create Program (CRTPGM) command, except that the module created by the Create Module command step is deleted after the CRTPGM step.

To use the Create Bound Program commands, the source member must contain a `main()` function.

Note: When a CRTPGM parameter does not appear in the Create Bound Program command, the CRTPGM parameter default is used. For example, the parameter ACTGRP(*NEW) is the default for the CRTPGM command, and is used for the Create Bound Program command. You can change the CRTPGM parameter defaults by using the Change Command Defaults (CHGCMDDFT) command.

You can use the CRTSQLCI command to start the ILE C compiler and create a program object. The SQL database can be accessed from an ILE C program if you embed SQL statements in the ILE C source.

Example

1. To create the program T1520ALP, using the source found in Figure 4 on page 21, type the following command line:

```
CRTBND CPG(MYLIB/T1520ALP) SRCFILE(QCPPLE/QACSRC)
      TEXT('Adds integers and prints characters') OUTPUT(*PRINT)
      OPTION(*EXPMAC *SHOWINC *NOLOGMSG) FLAG(30) MSGLMT(10)
      CHECKOUT(*PARM) DBGVIEW(*ALL)
```

The options specified are:

- OUTPUT(*PRINT) - specifies that you want a compiler listing.
- OPTION(*EXPMAC *SHOWINC *NOLOGMSG) - specifies that you want to expand include files and macros in a compiler listing and not log messages in the job log.
- FLAG(30) - specifies that you want severity level 30 messages to appear in the listing.
- MSGLMT(10) — specifies that you want compilation to stop after 11 messages at severity level 30.
- CHECKOUT(*PARM) — shows a list of function parameters not used. DBGVIEW(*ALL) specifies that you want all three views and debug data to debug this program.

CLE, the program attribute, identifies this program as an Integrated Language Environment program.

2. Type one of the following CL commands to see the compiler listing:

- DSPJOB and then select option 4 (Display spooled files)
- WRKJOB and then select option 4 (Work with spooled files)
- WRKOUTQ *queue-name*
- WRKSPLF

Select an option to see the compiler listing.

3. To run the program type:

```
CALL PGM(MYLIB/T1520ALP)
```

4. Type a and press Enter. Type 9 and press Enter. Type b and press Enter. Type 8 and press Enter. Type + and press Enter.

The interactive session is as shown:

```
> a
> 9
> b
> 8
> +
Characters are ab
Sum of Digits is 17
Press ENTER to end terminal session.
```

Creating a Program in Two Steps

To take advantage of the flexibility that ILE C/C++ offers, you can compile and bind source code into an ILE C/C++ program in two steps:

1. In the first step, you create one or more ILE C/C++ **module objects** (*MODULE) from their respective source members using the Create Module command.
2. In the second step, you use the Create Program (CRTPGM) command to bind one or more of these module objects into an executable ILE **program object** (*PGM). Binding is the process of combining one or multiple modules and optional service programs, and resolving external symbols between them. The system code that combines modules and resolves symbols is called the binder.

For example,

```
CRTCMOD HELLO
CRTPGM HELLO
CALL HELLO
```

Using modules has these advantages:

- Modules are easier to maintain. It is easier to maintain a small module representing a single function than to maintain an entire program. For example, if you change only a line or two in a module, you may only need to recompile the module, rather than the entire program.
- Modules are easier to test. Testing of functions can be done in isolation. You do not have to run the entire program. A test harness which includes the module under test can be used instead.
- Modules are easier to code. You can subdivide the work into smaller source members rather than coding an entire program in a single source file.
- Modules can be reused in different application programs.

Identifying Program and User Entry Procedures

When a module object is created, a program entry procedure (PEP) and a user entry procedure (UEP) may also be generated.

Both ILE C and C++ require the `main()` function, but in ILE C, it becomes the **UEP** of an ILE program. After the PEP runs, it calls the associated UEP, and starts the Integrated Language Environment program running.

As part of the binding process, a procedure must be identified as the startup procedure, or program entry procedure (PEP). When a program is called, the PEP receives the command line parameters and is given initial control for the program. The procedures that get control from the PEP are called user entry procedures (UEP).

An ILE module contains a program entry procedure only if it contains a main() function. Therefore, one of the modules being bound into the program must contain a main() function.

Understanding the Internal Structure of a Program Object

Figure 5 shows the internal structure of a typical program object, MYPROG, created by binding two modules, TRNSRPT and INCALC. In this example, TRNSRPT is the entry module containing the PEP, in addition to a UEP. Module INCALC contains a UEP only.

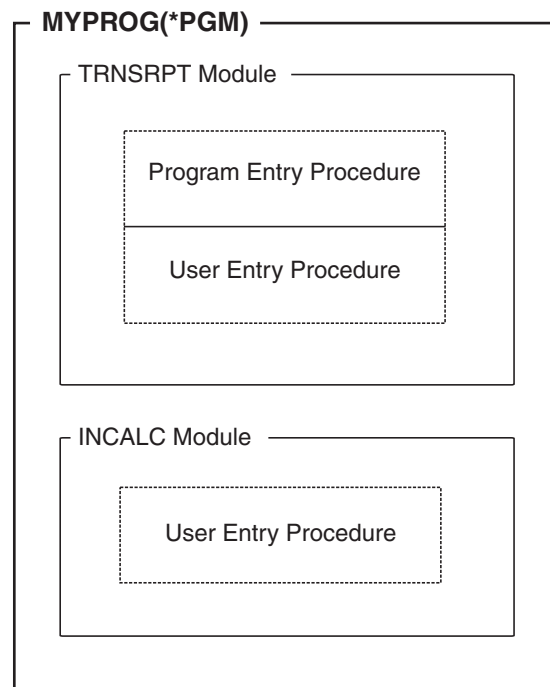


Figure 5. Structure of Program MYPROG

Using Static Procedure Calls

Within a bound object, procedures can be called using static procedure calls. These bound calls are faster than external calls. Therefore, an application consisting of a single bound program with many bound calls should perform faster than a similar application consisting of separate programs with many external inter-program calls.

Working with Binding Directories

A binding directory contains the names of the modules and service programs that you may need when creating an ILE program or service program.

Binding directories offer:

- A convenient method of packaging modules or service programs that you may need when creating an Integrated Language Environment program or service program.
- Reduce program size, because modules or service programs listed in a binding directory are used only if they are needed.

Binding directories are optional. They are objects identified to the system by the *BNDDIR* parameter on the CRTPGM command.

Modules or service programs listed in a binding directory are used only if they provide an export that can satisfy any currently unresolved import requests. Entries in the binding directory may refer to objects that do not yet exist at the time the binding directory is created, but exists later.

Creating a Binding Directory

If you want to create a binding directory, use the Create Binding Directory (CRTBNDDIR) command to contain the names of modules and service programs that your ILE C/C++ program or service programs may need.

For example,

```
CRTBNDDIR BNDDIR(MYBNDDIR) MODULES (MOD1, MOD2)
CRTCMOD PROG(MYPROG) BNDDIR (MYBNDDIR)
```

or

```
CRTBNDDIR BNDDIR(MYBNDDIR) MODULES (MOD1, MOD2)
CRTCPMOD PROG(MYPROG) BNDDIR (MYBNDDIR)
```

Using the Binder to Create a Program

The binder is invoked through the Create Program (CRTPGM) or the Create Service Program (CRTSRVPGM) commands. The CRTPGM command creates a program object from one or more module object objects and, if required, binds to one or more service programs. The CRTSRVPGM command creates a service program object from one or more module objects and, if required, binds to one or more service programs. See Chapter 3, “Service Programs” on page 33 for more information about service programs.

The CRTPGM and CRTSRVPGM commands invoke an OS/400® component referred to as the **binder**. The **binder** processes import requests for procedure names and data item names from specified modules. The binder then tries to find matching exports in the specified modules, service programs, and binding directories. An **export** is an external symbol defined in a module or program that is available for use by other modules or programs. An **import** is the use of, or reference to, the name of a procedure or data item that is not defined in the current module object.

You can bind modules created by the compiler with modules created by any of the other ILE Create Module commands, including CRTRPGMOD, CRTCMOD, CRTCLMOD, or CRTCLMOD, or other ILE compilers.

Note: The modules or service programs to be bound must already have been created.

Preparing to Create a Program

Before you create a program object using the CRTPGM command, you should:

1. Establish a program name.
2. Identify the module(s) and, if required, the service programs you want to bind into a program object.
3. Make sure that the program has a program entry procedure that gets control when a dynamic program call is made. (That is, one module must contain the `main()` function of the program.)

You indicate which module contains the program entry procedure through the *ENTMOD* parameter. The default is *ENTMOD(*FIRST)*, which means that the module containing the first program entry procedure found in the list for the *MODULE* parameter is the entry module.

If you are binding more than one ILE module together, you should specify *ENTMOD(*FIRST)* or else specify the module name with the program entry procedure. You can use *ENTMOD(*ONLY)* when you are binding only one module into a program object, or if you are binding several modules but only one contains a program entry procedure. For example, if you bind a module with a *main()* function to a C module without a *main()* function, you can specify *ENTMOD(*ONLY)*.

4. Identify the activation group that the program is to use.

Specify *ACTGRP(*NEW)* if your program has no special requirements or if you are not sure which group to use.

Note that *ACTGRP(*NEW)* is the default activation group for CRTPGM. This means that your program will run in its own activation group, and the activation group will terminate once the program terminates. This default ensures that your program has a refresh of the resources necessary to run, every time you call it.

See “Activation Groups” on page 30 for more information on unnamed and named activation groups.

Specifying Parameters for the CRTPGM Command

Table 3 lists CRTPGM command parameters and their default values. For a detailed description of the parameters, refer to the CL Reference CHKxxx through CVTxxx Commands: SC41–5724. Each parameter has default values which are used whenever you do not specify your own values.

Table 3. Parameters for CRTPGM Command and their Default Values

Parameter Group	Parameter(Default Value)
Identification	PGM(library name/program name) MODULE(*PGM) TEXT(*ENTMODTXT)
Program access	ENTMOD(*FIRST)
Binding	BNDSRVPGM(*NONE) BNDDIR(*NONE)
Run time	ACTGRP(*NEW)
Miscellaneous	OPTION(*GEN *NODUPPROC *NODUPVAR *WARN *RSLVREF) DETAIL(*NONE) ALWUPD(*YES) ALWLIBUPD(*NO) USRPRF(*USER) REPLACE(*YES) AUT(*LIBCRTAUT) TGTRLS(*CURRENT) ALWRINZ(*NO) STGMDL(*SNGLVL) IPA(*NO) IPACTLFILE(*NONE) IPARPLIL(*NO)

Resolving Import Requests

Once you have entered the CRTPGM command, the system does the following:

1. Copies listed modules into what will become the program object and links any service programs to the program object.
2. Identifies the module containing the program entry procedure and locates the first import in this module.
3. Checks the modules in the order in which they are listed and matches the first import with a module export.
4. Returns to the first module and locates the next import.
5. Resolves all imports in the first module.
6. Continues to the next module and resolves all imports.
7. Resolves all imports in each subsequent module until all of the imports have been resolved.
8. If any imports cannot be resolved with an export, terminates the binding process without creating a program object.
9. Once all the imports have been resolved, completes the binding process and creates the program object.

Handling Duplicate Variable Names

If you have specified in the binder language that a variable is to be exported (using the **EXPORT** keyword), it is possible that the variable name will be identical to a variable in another procedure within the bound program object.

Use the **DUPPROC* option on the CRTPGM *OPTION* parameter to allow duplicate procedure names. See *ILE C/C++ Compiler Reference* for further information on how to handle this situation.

Using a Binder Listing

The binding process can optionally produce a binder listing that describes the resources used, symbols and objects encountered, and problems that were resolved, or not resolved, in the binding process.

The listing is produced as a spooled file for the job you use to enter the CRTPGM command. You can choose a *DETAIL* parameter value to generate the listing at three levels of detail:

- **BASIC*
- **EXTENDED*
- **FULL*

The default is not to generate a listing. If it is generated, the binder listing includes the sections described in Table 4, depending on the value specified for *DETAIL*.

Table 4. Sections of the Binder Listing based on the *DETAIL* Parameter

Section Name	<i>*BASIC</i>	<i>*EXTENDED</i>	<i>*FULL</i>
Command Option Summary	X	X	X
Brief Summary Table	X	X	X
Extended Summary Table		X	X
Binder Information Listing		X	X
Cross-Reference Listing			X
Binding Statistics			X

The information in this listing can help you diagnose problems if the binding was not successful, or give feedback about what the binder encountered during the binding process.

Figure 6 shows the basic binder listing for a program CVTHEXPGM. Note that this listing is taken out of context. It only serves to illustrate the type of information you may find in a binder listing.

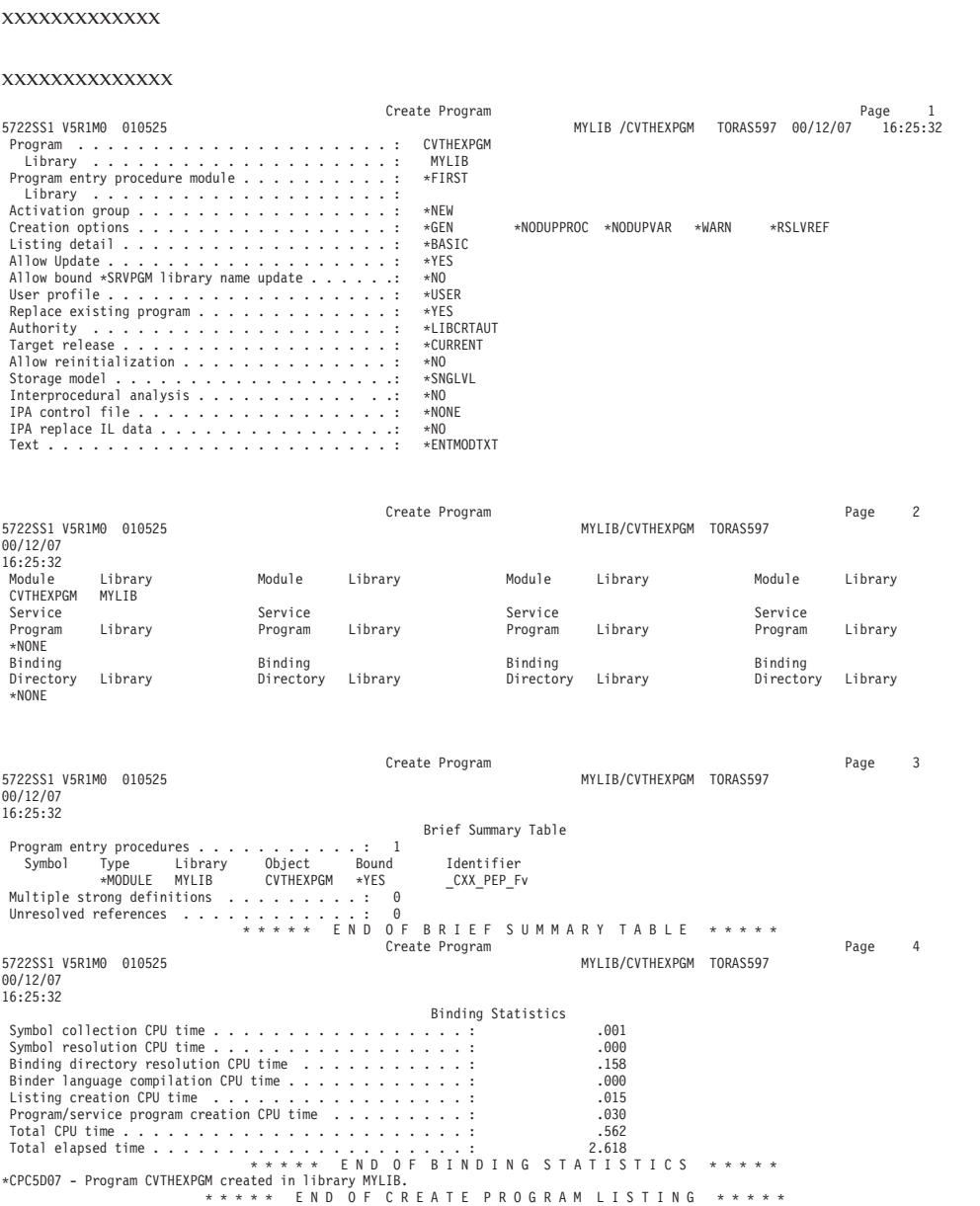


Figure 6. Example of a Basic Binder Listing

Updating a Module or a Program Object

There are many reasons why you may want to change a module or a program object:

- An object may need to be changed to accommodate enhancements, or for maintenance reasons.

You can isolate what needs to be changed by using debugging information or the binder listing from the CRTPGM command. From this information you can determine what modules, procedures, or fields need to change.

- You may want to change the optimization level or observability of a module or program.

This is often the case when you want to debug a program or module, or when you are ready to put a program into production. Such changes can be performed more quickly and use fewer system resources than the re-creation of the object in question.

- You may want to reduce the program size once you have completed an application.

ILE program objects have additional data added to them, which makes them larger than similar OPM or EPM program objects.

Each of the above approaches requires different data to make the change.

Updating a Program

In general, you can update a program by replacing modules as needed. You do not have to re-create the program object. The ability to replace specific modules is helpful if, for example, you are supplying an application to other sites that are already using the program. You need only send the revised modules, and the receiving site can update the application using the UPDPGM and UPDSRVPGM commands.

The update commands work with both program and module objects. The parameters for these commands are very similar to those for the Create Program (CRTPGM) command. For example, to replace a module in a program, you would enter the module name for the *MODULE* parameter and the library name.

To use the UPDPGM command, the modules to be replaced must be located in the same libraries they were in when the program was created. You can specify that all modules, or only some subsets of modules, are to be replaced.

Activation Groups

Activation is the process used to prepare an Integrated Language Environment program to run. Activation allocates and initializes static storage for an Integrated Language Environment program, and completes the binding of Integrated Language Environment programs to Integrated Language Environment service programs. The ACTGRP parameter on the CRTPGM and CRTSRVPGM commands specifies the activation group in which a program or service program runs.

All Integrated Language Environment programs and service programs are activated within a substructure of a job called an **activation group**. This substructure contains the resources necessary to run the Integrated Language Environment programs. The static and automatic program variables and dynamic storage are assigned separate address spaces for each activation group. Activation and activation groups:

- Help ensure that Integrated Language Environment programs running in the same job run independently without intruding on each other (for example, commitment control, overrides, shared files) by scoping resources to the activation group.
- Scope resources to the Integrated Language Environment program.
- Uniquely allocate the static data needed by the Integrated Language Environment program or service program.

- Change the symbolic links to ILE service programs into physical addresses.

Chapter 3. Service Programs

This chapter describes OS/400 service programs and how to create them.

A service program is an OS/400 object of type *SRVPGM. Service programs are typically used for common functions that are frequently called by other procedures within an application and across applications. For example, the ILE compilers use service programs to provide run-time services such as math functions and input/output routines.

Service programs simplify maintenance, and reduce storage requirements, because only a single copy of a service program is maintained and stored.

Differences between Programs and Service Programs

A service program differs from a program in two ways:

- A service program is bound to existing programs or other service programs. It cannot run independently.
- A service program does not contain a program entry procedure. Therefore, you cannot call a service program using an "OS" linkage specification. However, you can call a service program with a "c" linkage specification, because it contains at least one user entry procedure. A service program may have data exports rather than a user entry procedure.
- Service programs are bound by reference. This means that the content of the service program is not copied into the program to which it is bound. Instead, linkage information about the service program is bound into the program.

This process is different from the static binding process used to bind modules into programs. However, you can still call the service program's exported procedures as if they were statically bound. The initial activation is longer, but subsequent calls to any of the service program's exported procedures are faster than program calls.

Public Interface

The **public interface** of a service program consists of the names of the exported procedures and data items that can be referenced by other Integrated Language Environment objects. In order to be exported from an ILE service program, a data item must be exported from one of the module objects making up the ILE service program.

The exports list is used to specify the public interface for a service program. A **signature** is generated from the procedure and data item names listed in the binder language. This signature can then be used to validate the interface to the service program. As long as the public interface is unchanged, the clients of a service program do not have to be recompiled after a change to the service program.

Using the Binder to Create a Service Program

Creating a service program involves compiling source code into module objects, and then binding one or more module objects into a service program object with the Create Service Program (CRTSRVPGM) command. You can also use modules created with other ILE language compilers, such as ILE C/C++, ILE RPG/400, or ILE COBOL/400.

Considerations when Creating a Service Program

When creating a service program, you should consider:

- Whether or not you intend to update the program at a later date.
- Whether or not any updates involve changes to its interface.

If the interface to a service program changes, you may have to rebind all programs bound to the original service program. However, depending on the changes and how you implement them, you may be able to reduce the amount of rebinding if you create the service program using binder language. In this case, after updating the binder language source to identify new exports, you need to rebind only those programs that require the new exports.

Specifying Parameters for the CRTSRVPGM Command

Table 5 lists CRTSRVPGM command parameters and their default values.. For a detailed description of the parameters, refer to the CL Reference CHKxxx through CVTxxx Commands: SC41–5724. Each parameter has default values which are used whenever you do not specify your own values.

Table 5. Parameters and Default Values for CRTSRVPGM Command

Parameter Group	Parameter(Default Value)
Identification	SRVPGM(library name/service program name) MODULE(*SRVPGM)
Program access	EXPORT(*SRCFILE) SRCFILE(*LIBL/QSRVSRC) SRCMBR(*SRVPGM)
Binding	BNDSRVPGM(*NONE) BNDDIR(*NONE)
Run time	ACTGRP(*CALLER)
Miscellaneous	OPTION(*GEN *NODUPPROC *NODUPVAR *WARN *RSLVREF) DETAIL(*NONE) ALWUPD(*YES) ALWRINZ(*NO) REPLACE(*YES) AUT(*LIBCRTAUT) TEXT(*ENTMODTXT) TGTRLS(*CURRENT) USRPRF(*USER) STGMDL(*SNGLVL) IPA(*NO) IPACTLFILE(*NONE) IPARPLIL(*NO)

Updating or Changing a Service Program

You can update or change a service program in the same way you modify a program object. In other words, you can:

- Update the service program (using UPDSRVPGM)
- Change the optimization level (using CHGSRVPGM)
- Remove observability (using CHGSRVPGM)
- Reduce the size (using CPROBJ).

See “Updating a Module or a Program Object” on page 29 for more information on any of the above points.

If you use binder language, a service program can be updated without requiring programs calling it to be recompiled. For example, to add a new procedure to an existing service program:

1. Create a module object for the new procedure.
2. Modify the binder-language source file to handle the interface associated with the new procedure. Add any new export statements following the existing ones. See “Updating a Service Program Export List” on page 42 for details on modifying binder-language source files.
3. Recreate the original service program and include the new module.

Now existing programs can access the new functions. Since the old exports are in the same order, they can still be used by the existing programs. Until it is necessary to also update the existing programs, they do not have to be recompiled.

Using Related CL commands

The following CL commands can be used with service programs:

- Create Service Program (CRTSRVPGM)
- Change Service Program (CHGSRVPGM)
- Display Service Program (DSPSRVPGM)
- Delete Service Program (DLTSRVPGM)
- Update Service Program (UPDSRVPGM)
- Work with Service Program (WRKSRVPGM).

Creating a Sample Service Program

The following example shows how to create a service program SEARCH that can be called by other programs to locate a character string in any given string of characters.

Creating the Source Files

The SEARCH program is implemented as a class object Search. The class Search contains:

- Three private data members: `skippat`, `needle_p`, and `needle_size`
- Three constructors, each taking different arguments
- A destructor
- An overloaded function `where()`, which takes four different sets of arguments

The service program is composed of the following files:

- A user-defined header file `search.h`
- A source code file `search.cpp`
- A source code file `where.cpp`

User Header File

The class and function declarations are placed into a separate header file, `search.h`, shown below:

```
// header file search.h
// contains declarations for class Search, and inlined function
// definitions

#include <iostream.h>

class Search {
private:
    char skippat[256];
    char * needle_p;
    int  needle_size;
public:

    // Constructors
    Search( unsigned char * needle, int size);
    Search ( unsigned char * needle);
    Search ( char * needle);

    //Destructor
    ~Search () { delete needle_p;}

    //Overloaded member functions
    unsigned int where ( char * haystack) {
        return where (haystack, strlen(haystack));
    }
    unsigned int where ( unsigned char * haystack) {
        return where (haystack, strlen((const char *)haystack));
    }
    unsigned int where ( char * haystack, int size) {
        return where ( (unsigned char *) haystack, size);
    }
    unsigned int where ( unsigned char * haystack, int size);
};
```

Source Code Files

The definitions for the member functions of class `Search` that are not inlined in the class declaration are contained in two separate files: the source file `search.cpp`, which contains constructor definitions for class `Search`; and `where.cpp`, which contains the member function definition. These files are shown below:

```
// source file search.cpp
// contains the definitions for the constructors for class Search

#include "search.h"

Search::Search( unsigned char * needle, int size)
    : needle_size(size) , needle_p ( new char [size])
{
    memset (skippat, needle_size, 256);
    for (unsigned int i=0; i<size; ++i) {
        skippat [needle [i]] = size -i-1;
    }
    memcpy (needle_p, needle, needle_size);
}

Search::Search ( unsigned char * needle) {
    needle_size = strlen( (const char *)needle) ;
    needle_p = new char [needle_size];
    memset (skippat, needle_size, 256);
    for (unsigned int i=0; i<needle_size; ++i) {
        skippat [needle [i]] = needle_size -i-1;
    }
    memcpy(needle_p, needle, needle_size);
}
```



```

Search::Search ( char * needle) {
    needle_size = strlen( needle) ;
    needle_p = new char [needle_size];
    memset (skippat,needle_size, 256);
    for (unsigned int i=0; i<needle_size; ++i) {
        skippat [needle [i]] = needle_size -i-1;
    }
    memcpy(needle_p, needle, needle_size);
}

// where.cpp
// contains definition of overloaded member function for class Search

#include "search.h"

unsigned int Search:: where ( unsigned char * haystack, int size)
{ unsigned int i, t;
  int j;
  for ( i= needle_size-1, j = needle_size-1; j >= 0; --i, --j ){
      while ( haystack[i] != needle_p[j]) {
          t = skippat [ haystack [i]] ;
          i += (needle_size - j > t) ? needle_size - j : t ;
          if (i >= size)
              return size;
          j = needle_size - 1;
      }
  }
  return ++i;
}

```

The modules that result from the compilation of these source files, SEARCH and WHERE are bound into a service program, SERVICE1.

Compiling and Binding the Service Program

To create the service program SERVICE1, issue the following command:
 ctrsrvpgm srvgpm(mylib/service1) export(*all)

By default, the binder creates the service program in your current library.

The parameter *EXPORT(*ALL)* specifies that all data and procedures exported from the modules are also exported from the service program.

Binding the Service Program to a Program

In the following example, a very short application consisting of a program MYPROG is bound to the service program. The source code for MYPROG, myprog.cpp, is shown below.

Note: This sample application has been reduced to minimal functionality. It's main purpose is to demonstrate how to create a service program.

```

// myprog.cpp
// Finds a character string in another character string.

#include <stdio.h>
#include <iostream.h>
#include <stdlib.h>
#include "search.h"
#define HS "Find the needle in this haystack"

void main () {
    int i;

```

```

Search token("needle");
i = token.where (HS, sizeof(HS));
cout << "The string was found in position " << i << endl;
}

```

The program creates an object of class Search. It invokes the constructor with a value that represents the string of characters ("needle") to be searched for. It calls the member function where() with the string to be searched ("Find the needle in this haystack"). The string "needle" is located, and its position in the target string "Find a needle in this haystack" is returned and printed.

To create the program MYPROG in library MYLIB, and bind it to the service program SERVICE1, enter the following:

```
crtpgm pgm(mylib/myprog) bndsrvgm(mylib/service1) myprog.cpp
```

Figure 7 shows the internal and external function calls between program MYPROG and service program SERVICE1.

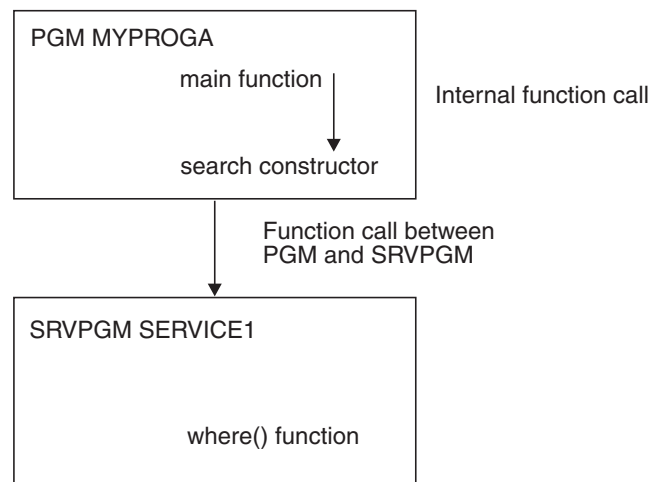


Figure 7. Calls between Program and Service Program

When MYPROG is created, it includes information regarding the interface it uses to interact with the service program.

To run the program, type:

```
call mylib/myprog
```

During the process of making MYPROG ready to run, the system verifies that:

- The service program SERVICE1 in library MYLIB can be found.
- The public interface used by MYPROG when it was created is still valid at run time.

If either of the above is not true, an error message is issued.

The output of MYPROG is:

```
The string was found in position 9
```

Chapter 4. Working With Exports From Service Programs

This section describes how to work with procedures and data items that can be exported from a service program.

Determining Exports from Service Programs

A service program exports procedures and data items that can be imported by other programs. These exports represent the interface to the service program. In the C/C++ programming language, procedures and data items correspond to functions and variables.

Information about exports, which can be derived from the modules that form a particular service program, may be used to create a binder language source file which then defines the interface to this service program. A binder language source file specifies the exports the service program makes available to all programs that call it. This file can be specified on the EXPORT parameter of the CRTSRVPGM command.

Binder language gives you better control over the exports of a service program. This control can be very useful if you want to:

- Determine export and import mismatches in an application.
- Add functionality to service programs.
- Reduce the impact of changes to a service program on the users of an application.
- Mask certain service program exports from service program users. That is, by not listing certain functions or variables in the binder language source file, you can prevent any calling programs from having access to these exports.

Displaying Export Symbols With the Display Module Command

To find out which exports are available from a module, type:

```
DSPMOD library-name/module-name
```

on a command line, indicating the module name and the library where the module is stored. This command brings up the Display Module Information display. At the bottom of this display, you find information about exported defined symbols, consisting of the name and type of each symbol that can be exported from the module.

Note: When the compiler compiles a source file, it encodes function names and certain variables to include type and scoping information. This encoding process is called name mangling. The symbol names in the sample display below are shown in mangled form. The source code for module SEARCH is shown in “Source Code Files” on page 36.

```

Display Module Information
Module . . . . . : SEARCH
Library . . . . . : MYLIB
Detail . . . . . : *EXPORT
Module attribute . . . . . :
Exported defined symbols:
Symbol Name                                     Symbol Type
__ct_6SearchFPc                                PROCEDURE
__ct_6SearchFPuc                                PROCEDURE
__ct_6SearchFPuci                              PROCEDURE

```

Figure 8. Display Module Information Screen for a Sample Module SEARCH

Creating a Binder Language Source File

Binder language is based on the exports available from modules that are bound into service programs. A binder language source file must contain the following entries:

1. The Start Program Export (STRPGMEXP) command identifies the beginning of the list of exports from the service program.
2. Export Symbol (EXPORT) commands identify each a symbol name available to be exported from the service program.
3. The End Program Export (ENDPGMEXP) command identifies the end of the list of exports from the service program.

The following example shows the structure of a binder language source file:

```

STRPGEXP PGMLEVEL(*CURRENT)
EXPORT SYMBOL("mangled_procedure_name_a")
EXPORT SYMBOL("mangled_procedure_name_b")
...
...
EXPORT SYMBOL("mangled_procedure_name_x")
ENDPGMEXP

```

Note: You must specify the mangled name of each symbol on the EXPORT command, because the binder looks for the mangled names of exports when it tries to resolve import requests from other modules.

Once all the modules to be bound into a service program have been created, you can create the binder language source file. You can write this file yourself, using the Source Entry Utility (SEU), or you can let the iSeries system generate it for you, through the Retrieve Binder Source (RTVBNDSRC) command.

Creating Binder Language Using SEU

You can use the Source Entry Utility (SEU) to create a binder language source file:

1. Create a source physical file QSRVSRC in library MYLIB.
2. Create a member MEMBER1 that will contain the binder language
3. Use the DSPMOD command to display the symbols that can be exported from each module.
4. Decide which exports you want to make available to calling programs.
5. Use the Source Entry Utility (SEU) to enter the syntax of the binder language.

You need one export statement for each procedure whose exports you want to make available to the caller of the service program. Do not list symbols that you do not want to make available to calling programs.

For example, based on the information shown in Figure 8 on page 40, the binder language source file for module SEARCH could list the following export symbols:

```
STRPGEXP PGMLEVEL(*CURRENT)
EXPORT SYMBOL(" __ct__6SearchFPc")
EXPORT SYMBOL(" __ct__6SearchFPUc")
EXPORT SYMBOL(" __ct__6SearchFPUci")
ENDPGMEXP
```

Creating Binder Language Using the RTVBNDSRC Command

The Retrieve Binder Source (RTVBNDSRC) command can automatically create a binder language source file. It retrieves the exports from a module, or a set of modules. It generates the binder language for these exports, and places exports and binder language in a specified file member. This file member can later be used as input to the *EXPORT* parameter of the CRTSRVPGM command.

Note: After the binder language has been retrieved into a source file member, you can edit the binder language and modify it as needed, for example, if you make changes to a module, or if you want to make certain exports unavailable to calling programs.

The syntax for the RTVBNDSRC command is: RTVBNDSRC
MODULE(MYLIB/SEARCH) SRCFILE(MYLIB/QSRVSRC) SRCMBR(*DFT) MBROPT(*REPLACE)

For detailed information on the RTVBNDSRC command and its parameters enter RTVBNDSRC on a command line and press F1 for Help.

Example of Creating Binder Language With RTVBNDSRC

The following example shows how to create a binder language source file for module SEARCH, located in library MYLIB, using the RTVBNDSRC command. The source code for module SEARCH is shown in “Source Code Files” on page 36.

```
RTVBNDSRC MODULE(MYLIB/SEARCH) SRCFILE(MYLIB/QSRVSRC) SRCMBR(ONE)
```

This command automatically:

1. Creates a source physical file QSRVSRC in library MYLIB.
2. Adds a member ONE to QSRVSRC.
3. Generates binder language from module SEARCH in library MYLIB and places it in member ONE.

Member ONE in file MYLIB/QSRVSRC now contains the following binder language:

```

Columns . . . :   1  71           Browse           MYLIB/QSRVSRG
SEU==>                                     ONE
FMT **   ...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7
***** Beginning of data *****
0000.01 STRPGMEXP PGMVLV(*CURRENT)
0000.02 /*****
0000.03 /*  *MODULE      SEARCH      MYLIB      95/06/10  17:34:41      */
0000.04 /*****
0000.05  EXPORT SYMBOL("_ct_6SearchFPc")
0000.06  EXPORT SYMBOL("_ct_6SearchFPuc")
0000.07  EXPORT SYMBOL("_ct_6SearchFPuci")
0000.08 ENDPGMEXP
***** End of data *****

```

Figure 9. Binder Language Source File Generated for Module SEARCH

Updating a Service Program Export List

You can use binder language to reflect changes in the list of exports a service program makes available. When you create binder language, a signature is generated from the order in which the modules that form a service program are processed, and from the order in which symbols are exported from these modules. The **EXPORT** keyword in the binder language identifies the procedure and data item names that make up the signature for the service program.

When you make changes to the exports of a service program this does not necessarily mean that all programs that call this service program must be re-created. You can implement changes in the binder language such that they are backward compatible. Backward-compatible means that programs which depend on exports that remain unchanged do not need to be re-created.

To ensure backward compatibility, add new procedure or data item names to the end of the export list, and recreate the service program with the same signature. This lets existing programs still use the service program, because the order of the unchanged exports remains the same.

Note: When changes to a service program result in a loss of exports, or in a change of existing exports, it becomes difficult to update the export list without affecting existing programs that require its services. Changes in the order, number, or name of exports result in a new signature that requires the re-creation of all programs and service programs that use the changed service program.

Using the Demangling Functions

You can retrieve the mangled names of exported symbols with the RTVBNDSRC command. To help you find the corresponding demangled names, the runtime library contains a small class hierarchy of functions that you can use to demangle names and examine the resulting parts of the name. The interface is documented in the `<demangle.h>` header file.

Using the demangling functions, you can write programs to convert a mangled name to a demangled name and to determine characteristics of that name, such as its type qualifiers or scope. For example, given the mangled name of a function, the program returns the demangled name of the function and the names of its

qualifiers. If the mangled name refers to a class member, you can determine if it is static, const, or volatile. You can also get the whole text of the mangled name.

To demangle a name, which is represented as a character array, create a dynamic instance of the Name class and provide the character string to the class's constructor. For example, to demangle the name f__1XFi, create:

```
char *rest;  
Name *name = Demangle("f__1XFi", rest);
```

The demangling functions classify names into five categories: function names, member function names, special names, class names, and member variable names. After you construct an instance of class Name, you can use the Kind member function of Name to determine what kind of Name the instance is. Based on the kind of name returned, you can ask for the text of the different parts of the name or of the entire name.

For the mangled name f__1XFi, you can determine:

```
name->Kind() == MemberFunction  
((MemberFunctionName *) name)->Scope()->Text() is "X"  
((MemberFunctionName *) name)->RootName() is "f"  
((MemberFunctionName *) name)->Text() is "X::f(int)"
```

If the character string passed to the Name constructor is not a mangled name, the Demangle function returns NULL.

For further details about the demangling functions, refer to the information contained in the <demangle.h> header file. If you installed ILE C/C++ using default settings, this header file should be in IFS in the '/QIBM/include' directory and in DM in 'QSYSINC/H'.

Handling Unresolved Import Requests During Program Creation

An unresolved import is an import whose type and name do not yet match the type and name of an export. Unresolved import requests do not necessarily prevent you from creating a program or a service program. You can proceed in two ways:

- Specify the *UNRSLVREF option on the CRTPGM or CRTSRVPGM commands to tell the binder to go ahead and create a program or service program, even if there are imports in the modules, and no matching exports can be found.
- Change the order of program creation to avoid unresolved references.

Both approaches are demonstrated in "Creating a Program with Circular References" on page 44.

Use the *UNRSLVREF option to convert, create, or build pieces of code when all the pieces of code are not yet available. After the development or conversion phase has finished and all import requests can be resolved, make sure you re-create the program or service program that has the unresolved imports.

If you use the *UNRSLVREF option, specify *DETAIL(*EXTENDED)* or *DETAIL(*FULL)*, or keep the job log when the object is created, to identify the procedure or data item names that are not found.

Note: If you have specified *UNRSLVREF and a program is created with unresolved import requests, you receive an error message (MCH3203) when you try to run the program.

Creating a Service Program Using Binder Language

To create the service program described in “Creating a Sample Service Program” on page 35 using binder language, follow these steps:

1. To create modules from all source files enter the following compiler invocation:

```
CRTCPPMOD MODULE(MYLIB/SEARCH) SRCFILE(MYLIB/QCPPSRC)
```

The /C compiler option allows you to stop the compilation process after the creation of the modules SEARCH and QCPPSRC. The binder is not invoked.

2. To create the corresponding binder language source file, enter the command:

```
rtvbndsrc module(mylib/search mylib/qcppsrc)
srcfile(mylib/qsrvsrsrc) srcmbr(two)
```

This command creates the binder language source file shown in Figure 10.

3. To create service program SERVICE2, invoke the binder as follows:

```
crtsrvgpm srvgpm(mylib/service2) srcfile(mylib/qsrvsrsrc)
srcmbr(two)" search where
```

```
Columns . . . : 1 71          Browse          MYLIB/QSRVSRC
SEU==> TWO
FMT **  ...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7
          ***** Beginning of data *****
0000.01 STRPGMEXP PGMLVL(*CURRENT)
0000.02 /*****
0000.03 /*  *MODULE      SEARCH      MYLIB      95/06/11 15:30:51*/
0000.04 /*****
0000.05 EXPORT SYMBOL("_ct_6SearchFPc")
0000.06 EXPORT SYMBOL("_ct_6SearchFPUc")
0000.07 EXPORT SYMBOL("_ct_6SearchFPUci")
0000.08 /*****
0000.09 /*  *MODULE      WHERE      MYLIB      95/06/11 15:30:51*/
0000.10 /*****
0000.11 EXPORT SYMBOL("where__6SearchFPUci")
0000.12 ENDPGMEXP
          ***** End of data *****
```

Figure 10. Binder Language Source File Generated by the RTVBNDSRC Command

Creating a Program with Circular References

A **circular reference** is a special case of unresolved import requests. It occurs, for example, when a service program SP1 depends on imports from a service program SP2, which in turn depends on an import from service program SP1.

Figure 11 on page 45 illustrates the unresolved import requests between program A and two service programs, SP1 and SP2.

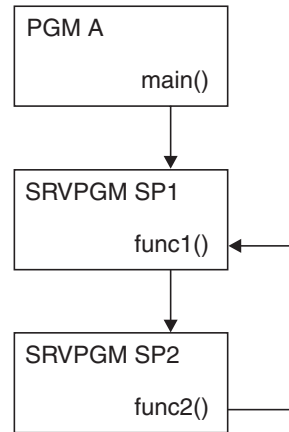


Figure 11. Unresolved Import Requests in a Program With Circular References

The following import requests occur between program A and the two service programs, SP1 and SP2, that are called by A:

1. Program A uses function `func1()`, which it imports from service program SP1.
2. Service program SP1 needs to import function `func2()` provided by service program SP2, in order to provide `func1()` to program A.
3. Service program SP2, in turn, first needs to import `func1` from service program SP1 before being able to provide `func2`.

Creating the Source Files

The application consists of three source files, `m1.cpp`, `m2.cpp`, and `m3.cpp`, shown below:

```

// m1.cpp
#include <iostream.h>
int main(void)
{
    void func1(int);
    int n = 0;
    func1(n);           // Function func1() is called.
}

// m2.cpp
#include <iostream.h>
void func2 (int);
void func1(int x)
{
    if (x<5)
    {
        x += 1;
        cout << "This is from func1(), n=" << x << endl;
        func2(x);       // Function func2() is called.
    }
}

// m3.cpp
#include <iostream.h>
void func1(int);
void func2(int y)
{
    if (y<5)
    {
        y += 1;
    }
}
  
```

```

        cout << "This is from func2(), n=" << y << endl;
        func1(y);          // Function func1() is called.
    }
}

```

Creating Modules

Compile the source files `m2.cpp` and `m3.cpp` into module objects from which you later create the service programs `SP1` and `SP2`. This allows you to display their exports with the `DSPMOD` command, or to generate binder language source with the `RTVBNDSRC` command. To create module objects from the source files described above, invoke the command:

```

CRTCPPMOD MODULE(MYLIB/m2) SRCFILE(MYLIB/QCPPSRC)
CRTCPPMOD MODULE(MYLIB/m3) SRCFILE(MYLIB/QCPPSRC)

```

The `CRTCPPMOD` compiler option indicates to the compiler that you do not want to create a program object from the source files. The target library is specified by the `MODULE` option.

Creating Binder Language

To generate binder language for module `M2`, from which you want to create service program `SP1`, issue the following command:

```
rtvbndsrc module(mylib/m2) srcfile(mylib/qsrvsrcc) srcmbr(bndlang1)
```

This command results in the following binder language being created for module `M2`, in library `MYLIB`, source file `QSRVSRCC`, file member `BNDLANG1`:

```

Columns . . . : 1 71          Browse          MYLIB/QSRVSRCC
SEU==>          BNDLANG1
FMT **  ...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7
          ***** Beginning of data *****
0000.01 STRPGMEXP PGMLVL(*CURRENT)
0000.02 /*****
0000.03 /*  *MODULE      M2          MYLIB      95/06/11 18:07:04*/
0000.04 /*****
0000.05 EXPORT SYMBOL("func1__Fi")
0000.06 ENDPGMEXP
          ***** End of data *****

```

Figure 12. Binder Language for Service Program SP1

To generate binder language for module `M3`, from which you want to create service program `SP2`, issue the following command:

```
rtvbndsrc module(mylib/m3) srcfile(mylib/qsrvsrcc) srcmbr(bndlang2)
```

This command results in the following binder language being created for module `M3`, in library `MYLIB`, source file `QSRVSRCC`, file member `BNDLANG2`:

```

Columns . . . :   1  71           Browse           MYLIB/QSRVSRC
SEU==>           BNDLANG2
FMT **   ...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7
          ***** Beginning of data *****
0000.01 STRPGMEXP PGMLVL(*CURRENT)
0000.02 /*****
0000.03 /*  *MODULE      M3          MYLIB      95/06/11 18:08:14  */
0000.04 /*****
0000.05 EXPORT SYMBOL("func2__Fi")
0000.06 ENDPGMEXP
          ***** End of data *****

```

Figure 13. Binder Language for Service Program SP2

Creating the Program

Program A will be created from m1.cpp. Service program SP1 will be created from M2. Service program SP2 is created from M3.

If you try and create service program SP1 from module M2, using the binder language shown in Figure 12 on page 46 and the compiler invocation:

```

CRTSRVPGM SRVPGM(MYLIB/SP1)
  SRCFILE(MYLIB/QSRVSRC) SRCMBR(BNDLANG1) m2

```

you find that the binder tries to resolve the import for function func2(), but fails, because it is not able to find a matching export. Therefore, service program SP1 is not created.

If SP1 is not created, this leads to problems if you try and create service program SP2 from module M3 using the binder language shown in Figure 13 and the compiler invocation:

```

CRTSRVPGM SRVPGM(MYLIB/SP2)
  SRCFILE(MYLIB/QSRVSRC) SRCMBR(BNDLANG2) m3

```

Service program SP2 is not created, because the binder fails in searching for the import for func1() in service program SP1, which has not been created in the previous step.

If you try and create program A with the compiler invocation:

```

crtpgm pgm (A) bndsrvpgm(MYLIB/SP1 MYLIB/SP2) m1.cpp

```

the binder fails, since service programs SP1 and SP2 do not exist.

Handling Unresolved Import Requests with *UNRSLVREF

The following scenario shows how to use the parameter *UNRSLVREF to handle the unresolved import requests which would otherwise prevent you from creating program A.

1. To create service program SP1 from m2.cpp, type:

```

CRTSRVPGM SRVPGM(MYLIB/SP1) SRCFILE(MYLIB/QSRVSRC)
  SRCMBR(BNDLANG1) OPTION(*UNRSLVREF) m2.cpp

```

Since the *UNRSLVREF option is specified, service program SP1 is created even though the import request for func2() is not resolved.

2. To create service program SP2 from module M3, type:

```
CRTSRVPGM SRVPGM(MYLIB/SP2) SRCFILE(MYLIB/QSRVSRRC)
SRCMBR(BNDLANG2) OPTION(*UNRSLVREF)" m3.cpp
```

Since service program SP1 now exists, the binder resolves all the import requests required, and service program SP2 is created successfully.

3. To re-create the service program SP1, type:

```
CRTSRVPGM SRVPGM(MYLIB/SP1) SRCFILE(MYLIB/QSRVSRRC)
SRCMBR(BNDLANG1) BNDSRVPGM(MYLIB/SP2) m2
```

Although service program SP1 does exist, the import request for func2() is not resolved. Therefore, the re-creation of service program SP1 is required. Since service program SP2 now exists, the binder resolves all import requests required and, service program SP1 is created successfully.

4. To create program A, type:

```
CRTPGM PGM(MYLIB/A) BNDSRVPGM(MYLIB/SP1 MYLIB/SP2)
```

Since service programs SP1 and SP2 do exist, the binder creates the program A.

Handling Unresolved Import Requests by Changing Program Creation Order

You can also change the order of program creation to avoid unresolved references, by first creating a service program with all modules, and then re-creating this same service program later.

1. To generate binder language for modules M2 and M3, from which you want to create service program SP1, issue the following command:

```
RTVBNDSRC MODULE(MYLIB/M2 MYLIB/M3) SRCFILE(MYLIB/QSRVSRRC)
SRCMBR(BNDLANG3)
```

This command results in the binder language shown in *Figure 14 on page 49* being created in library MYLIB, source file QSRVSRRC, file member BNDLANG3.

2. To create service program SP1 from module M2 and module M3 type:

```
CRTSRVPGM SRVPGM(MYLIB/SP1) SRCFILE(MYLIB/QSRVSRRC)
SRCMBR(BNDLANG3) m2.cpp m3.cpp
```

Since modules M2 and M3 are specified, all import requests are resolved, and service program SP1 is created successfully.

3. To create service program SP2, type:

```
CRTSRVPGM SRVPGM(MYLIB/SP2) SRCFILE(MYLIB/QSRVSRRC)
SRCMBR(BNDLANG2) BNDSRVPGM(MYLIB/SP1) m3.cpp
```

Since service program SP1 exists, the binder resolves all the import requests required and service program SP2 is created successfully.

4. To re-create service program SP1, type:

```
CRTSRVPGM SRVPGM(MYLIB/SP1) SRCFILE(MYLIB/QSRVSRRC)
SRCMBR(BNDLANG1) BNDSRVPGM(MYLIB/SP2) m2.cpp
```

Although service program SP1 does exist, the import request for func2() is not resolved to the one in service program SP2. Therefore, a re-creation of service program SP1 is necessary to make the circular reference work.

Since service program SP2 now exists, the binder can resolve the import request for func2() from service program SP2, and service program SP1 is successfully created.

- To create program A, type:

```
CRTPGM PGM(MYLIB/A) BNDSRVPGM(MYLIB/SP1 MYLIB/SP2) m1.cpp
```

Since service programs SP1 and SP2 do exist, the binder creates program A.

```
Columns . . . : 1 71          Browse          MYLIB/QSRVSRC
SEU==>          BNDLANG3
FMT **  ...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7
          ***** Beginning of data *****
0000.01 STRPGMEXP PGMLVL(*CURRENT)
0000.02 /*****
0000.03 /*  *MODULE      M2          MYLIB      95/06/11 18:50:23  */
0000.04 /*****
0000.05  EXPORT SYMBOL("func1__Fi")
0000.06 /*****
0000.07 /*  *MODULE      M3          MYLIB      95/06/11 18:50:23  */
0000.08 /*****
0000.09  EXPORT SYMBOL("func2__Fi")
0000.10 ENDPGMEXP
          ***** End of data *****
```

Figure 14. Binder Language for Service Program SP1

Binding a Program to a Non-Existent Service Program

To successfully create a program or a service program, all required modules must exist prior to invoking the binder.

However, if you want to bind a program to a non-existent service program, you can create a "placeholder" service program first. Consider the following example:

A program MYPROG requires a function myprint() to be exported by a service program PRINT. The code for the program is available in myprog.cpp. However, the source for the service program does not yet exist. To work around this problem:

- Create a source file dummy.cpp, such as:

```
//dummy.cpp
#include <iostream.h>
void function(void) {
    cout << "I am a placeholder only" << endl;
    return;
}
```

- Compile and bind dummy.cpp into a service program PRINT:

```
crtsrvgpm srvgpm(mylib/print) dummy.cpp
```

- Create the source file for program MYPROG:

```
// myprog.cpp
#include <iostream.h>
#define size 80
void print(char *);
main() {
    char text[size];
    cout << "Enter text" << endl;
    cin >> text;
    print(text);
    return;
}
```

- Create the program MYPROG from myprog.cpp and bind it to the service program PRINT:

```
crtpgm pgm(mylib/myprog) bndsrvgm(mylib/print)
option(*unrslvref) myprog.cpp
```

The option `*UNRSLVREF` ensures that the program binds to the service program, although there is no matching export for MYPROG's import void `print(char *)`.

Before you can run program MYPROG successfully, you must re-create service program PRINT from the real source code, instead of from the placeholder code in `dummy.cpp`.

Note: MYPROG only runs successfully if PRINT actually exports a function that matches MYPROG's import request.

Updating a Service Program Export List

To make backward-compatible changes to an ILE C/C++ service program you use the binder language. This language allows you to define a list of procedure names and data item names that can be exported. The `EXPORT` symbol (`EXPORT`) command in the binder language identifies the procedure and data item names that make up the signature for the service program module.

New procedure or data item names should be added to the end of the export list to ensure changes are compatible. A signature is generated by the order in which the modules are processed and the order in which the symbols are exported from the copied modules. A service program becomes difficult to update once the exports are used by other ILE C/C++ programs. If the service program is changed, the order or number of exports could change. If the signature changes all ILE C/C++ programs and service programs that use the changed service program have to be re-created.

The following example shows how to add a new procedure called `cost2()` to service program `COST` without having to re-create the existing program `COSTDPT1` that requires an export from `COST`.

Program Description

The figure below shows the exports in the existing version of service program `COST`, and in the updated version.

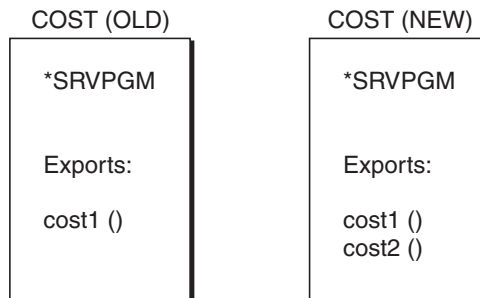


Figure 15. Exports from Service Program `COST`

The figure below shows the import requests in the existing program `COSTDPT1`, and in the new program `COSTDPT2`.

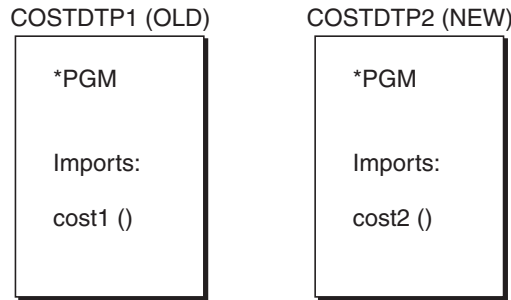


Figure 16. Import requests in Programs COSTDPT1 and COSTDPT2

The binder language for the old version of service program COST is located in member BND of source file QSRVSR, in library MYLIB:

```
STRPGMEXP PGMLVL(*CURRENT)
  EXPORT SYMBOL("cost1__Fi9_DecimalTXSP10SP2_")
ENDPGMEXP
```

The export signature is 94898385315FD06BB65E44D38A852904.

The updated binder language includes the new export procedure cost2(). It is located in member BNDUPD of source file QSRVSR, in library MYLIB:

```
STRPGMEXP PGMLVL(*CURRENT)
  EXPORT SYMBOL("cost1__Fi9_DecimalTXSP10SP2_")
  EXPORT SYMBOL("cost2__Fi9_DecimalTXSP10SP2_9_DecimalTXSP3SP1_")
ENDPGMEXP
```

The new export signature is 61E595C21D3EC9FDFD29749FB36B42D0.

In the binder language source that defines the old service program, the *PGMLVL* value is changed from **CURRENT* to **PRV*:

```
STRPGMEXP PGMLVL(*PRV)
  EXPORT SYMBOL("cost1__Fi9_DecimalTXSP10SP2_")
ENDPGMEXP
```

Its export signature is unchanged.

Note: If you want to ensure that existing programs can call the new version of the service program without being re-created, make sure to:

1. Add the new exports to the end of the symbol list in the binder language
2. Explicitly specify a signature for the new version of the service program that is identical to the signature of the old version.

Creating the Source Files

The source code for service program COST, module COST2, and programs COSTDPT1 and COSTDPT2 is shown below:

```
// cost1.cpp
// contains the export function cost1() for the old service program
#include <iostream.h>
#include <bcd.h>
_DecimalT<10,2> cost1 (
  int q,          // The quantity.
  _DecimalT<10,2> p ) // The price.
{
```

```

        _DecimalT<10,2> c;          // The cost.
        c = q*p;
        return c;
    }

// cost2.cpp
// contains the export function cost2() for the new service program
#include <iostream.h>
#include <bcd.h>
_DecimalT<10,2> cost2 (int quantity, _DecimalT<10,2> price,
                        _DecimalT<3,1> discount )
{
    _DecimalT<10,2> c = __D(quantity*price*discount/100);
    return c;
}

// costdpt1.cpp
// This program prompts users (from dept1) to enter the
// quantity, and price for a product. It uses function
// cost1() to calculate the cost, and prints the result out.
#include <iostream.h>
#include <bcd.h>
_DecimalT<10,2> cost1(int, _DecimalT<10,2>);
int main(void)
{
    int            quantity;
    _DecimalT<10,2> cost;
    _DecimalT<10,2> price;
    cout << "Enter the quantity, please." << endl;
    cin  >> quantity;
    cout << "Enter the price, please." << endl;
    cin  >> price;
    cost = cost1(quantity, price);
    cout << "The cost is $" << cost << endl;
}

// costdpt2.cpp
// This program prompts users (from dept2) to enter the
// quantity, price, and discount rate for a product.
// It uses function cost2() to calculate the cost, and prints
// the result out.
#include <iostream.h>
#include <decimal.h>
_DecimalT<10,2> cost2(int, _DecimalT<10,2>, _DecimalT<3,1>);
int main(void)
{
    int            quantity;
    _DecimalT<10,2> price;
    _DecimalT<10,2> cost;
    _DecimalT<3,1> discount;
    cout << "Enter the quantity, please." << endl;
    cin  >> quantity;
    cout << "Enter the price, please." << endl;
    cin  >> price;
    cout << "Enter the discount, please.( %)" << endl;
    cin  >> discount;
    cost = cost2(quantity, price, discount);
    cout << "The cost is be $" << cost << endl;
}

```

Compiling and Binding Programs and Service Programs

1. Create service program COST from source file cost1.cpp, using the binder source member BND, located in source file QSRVSRC, in library MYLIB:

```

CRTSRVPGM SRVPGM(MYLIB/COST) SRCFILE(MYLIB/QSRVSRC)
SRCMBR(BND) DETAIL(*EXTENDED) cost1.cpp

```


2. Create program COSTDPT1 from source file costdpt1.cpp and service program COST, located in library MYLIB:
`CRTPGM PGM(MYLIB/COSTDPT1) BNDSRVPGM(MYLIB/COST) costdpt1.cpp`
3. Update service program COST to include module COST2, using the updated binder language source BNDUPD, located in source file QSRVSR in library MYLIB:
`CRTSRVPGM SRVPGM(MYLIB/COST) SRCFILE(MYLIB/QSRVSR)
SRCMBR(BNDUPD) DETAIL(*EXTENDED) cost1 cost2.cpp`

It is necessary to re-create the service program COST, using the two modules COST1 and COST2 and the updated version of the binder language BNDUPD, so that it supports the new `cost2()` function. Program COSTDPT1, which used COST before it was re-created, remains unchanged.

In order to update service program COST, it is necessary to re-create it from the two modules COST1 and COST2, using the updated version of the binder language BNDUPD. The **EXTENDED* option in the *DETAIL* parameter creates an extended output listing, so that you can look at the current and previous signature of COST.

4. Create program COSTDPT2 from source file costdpt2:
`CRTPGM PGM(MYLIB/COSTDPT2) BNDSRVPGM(MYLIB/COST) costdpt2.cpp`

Running the Program

Run program COSTDPT1 from an OS/400 command line using the CL command `CALL COSTDPT1`.

Run program COSTDPT2 from an OS/400 command line using the CL command `CALL COSTDPT2`.

Chapter 5. Example - Creating a Sample ILE C Application

The following example demonstrates some typical steps in creating a sample ILE C application. The application is a small transaction-processing program that takes item names, price, and quantity as input. As output, the application displays the total cost of the items on the screen and writes an audit trail of the transaction.

Sample Application Overview

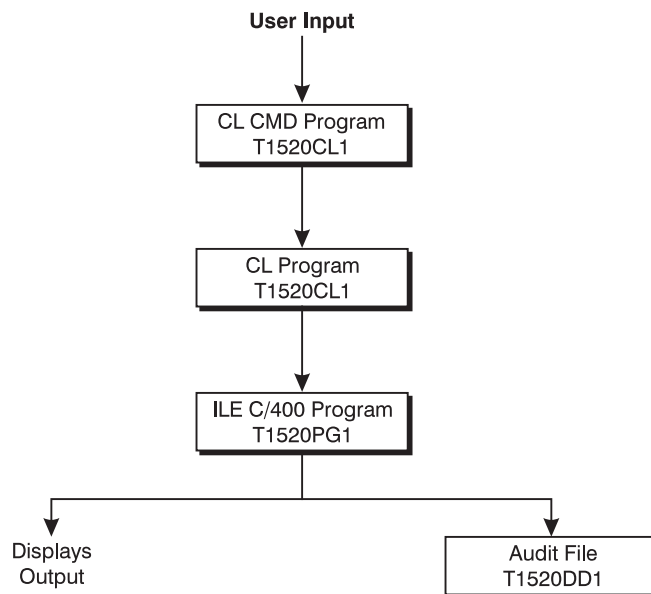


Figure 17. Basic Application Structure

This example consists of:

- A CL command that accepts the user's input, and passes it to a CL program.
- A CL program that processes the input, and passes it to an ILE C program.
- An ILE C program that processes the input and directs output to the user's terminal and to an externally described file. The ILE C program consists of two modules and each service program consists of a single module.
- A binder language source member.

In addition to the source for the CMD, CL, and ILE C programs, you need Data Description Specification (DDS) source for the output file, binder language source for the ILE C service programs and a binding directory for the ILE C service programs.

Note: This example is for illustration purposes. It is not necessary to create a binding directory for an ILE C program of this size. You might not want to break up ILE C service programs by data and function as shown.

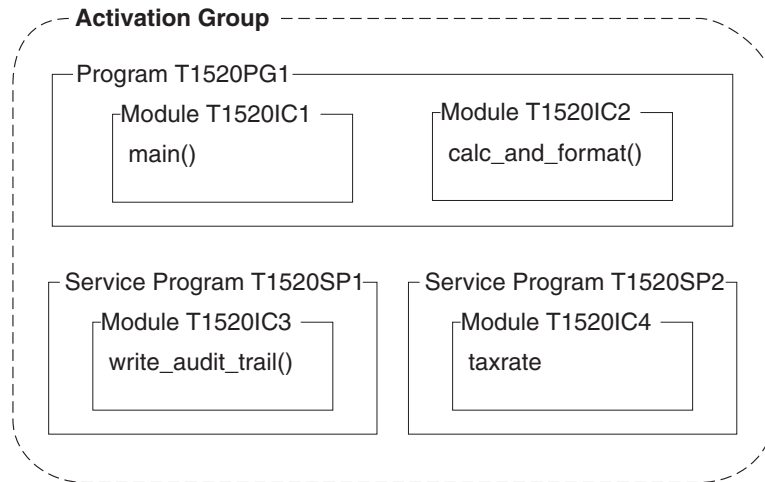


Figure 18. Structure of the Application in ILE C

When an OS/400 job starts, the system creates two activation groups to be used by OPM programs. One activation group is reserved for OS/400 system code. The other is used for all other OPM programs. You cannot delete the OPM default activation groups. They are deleted by the system when your job ends. An ILE C program may leave an activation group, and return to a call stack entry that is running in another activation group if any of the following are processed:

- An `exit()` function
- An unhandled exception
- A return statement from the `main()` function
- `Alongjmp()` function — used to jump back to a previous invocation that is before the previous control boundary

Therefore when the CL Command program calls the CL program, all the resources necessary to run these programs are allocated in the default activation group. When the CL program calls the ILE C program, because the ILE C program is created with `ACTGRP(*NEW)`, a new activation group is started. The ILE C service programs are also activated in this new activation group because they are created with `ACTGRP(*CALLER)`.

In the following example, the ILE C program and ILE C service programs are activated within one activation group because the programs are developed as one cooperative application. To isolate programs from other programs running in the same job you can select different activation groups. For example, a complete customer solution may be provided by integrating software packages from four different vendors. Different activation groups ease the integration by isolating the resources associated with each vendor package.

Task Overview

In the steps that follow, you will:

- Create a physical file to contain an audit trail for the ILE C program
- Create a CL program that passes the parameters item name, price, quantity, and user ID to an ILE C program.
- Create a CL command prompt to enter data for item name, price, and quantity. The CL command prompt passes the data to the CL program which in turn calls an ILE C program.

- Create one module with a `main()` function that receives the user ID, item name, quantity, and price from a CL program and calls `calc_and_format()` in one module to calculate an item's total cost, and the `write_audit_trail()` function in another module to write the transaction to an audit file.
- Create one module with `calc_and_format()` that completes the tax calculation by receiving arguments from the ILE C program and by importing the tax rate data item from an ILE C service program. The function `calc_and_format()` also formats the total cost.
- Create one module with `write_audit_trail()` that writes the audit trail for the ILE C program by importing the tax rate data item from an ILE C service program. This module creates an ILE C service program.
- Create one module that exports the tax rate data. This module creates an ILE C service program.
- Create a binding directory.
- Add two service program names to a binding directory.
- Create a source physical file to contain the binder language source to export a data item named `taxrate`.
- Create a source physical file to contain the binder language source to export a procedure named `write_audit_trail`.
- Create an ILE C service program from a module that exports the tax rate data. A source physical file containing the binder language source is also used.
- Create an ILE C service program from a module that exports a procedure called `write_audit_trail`. A source physical file containing the binder language source is also used.
- Create an ILE C program from two modules and two ILE C service programs.

Instructions

1. To create a physical file T1520DD1 using the source shown below, type:

```
CRTPF FILE(MYLIB/T1520DD1) SRCFILE(QCPPL/QADDSSRC) MAXMBRS(*NOMAX)
```

```
R T1520DD1R
```

USER	10	COLHDG('User')
ITEM	20	COLHDG('Item name')
PRICE	10S 2	COLHDG('Unit price')
QTY	4S	COLHDG('Number of items')
TXRATE	2S 2	COLHDG('Current tax rate')
TOTAL	21	COLHDG('Total cost')
DATE	6	COLHDG('Transaction date')

```
K USER
```

Figure 19. T1520DD1 — DDS Source for an Audit File

This file contains the audit trail for the ILE C program T1520PG1. The DDS source defines the fields for the audit file.

2. To create a CL program T1520CL1 using the source shown below, type:

```
CRTCLPGM PGM(MYLIB/T1520CL1) SRCFILE(QCPPL/QACLSRC)
```

```

PGM      PARM(&ITEMIN &PRICE &QUANTITY)
        DCL      VAR(&USER)      TYPE(*CHAR) LEN(10)
        DCL      VAR(&ITEMIN)    TYPE(*CHAR) LEN(20)
        DCL      VAR(&ITEMOUT)   TYPE(*CHAR) LEN(21)
        DCL      VAR(&PRICE)     TYPE(*DEC)  LEN(10 2)
        DCL      VAR(&QUANTITY)  TYPE(*DEC)  LEN(2 0)
        DCL      VAR(&NULL)      TYPE(*CHAR) LEN(1) VALUE(X'00')
        /* ADD NULL TERMINATOR FOR THE ILE C PROGRAM */
        CHGVAR   VAR(&ITEMOUT) VALUE(&ITEMIN *TCAT &NULL)
        /* GET THE USERID FOR THE AUDIT TRAIL */
        RTVJOBA  USER(&USER)
        /* ENSURE AUDIT RECORDS WRITTEN TO CORRECT AUDIT FILE MEMBER */
        OVRDBF   FILE(T1520DD1) TOFILE(*LIBL/T1520DD1) +
                MBR(T1520DD1) OVRSCOPE(*CALLLVL) SHARE(*NO)
        CALL     PGM(T1520PG1) PARM(&ITEMOUT &PRICE &QUANTITY +
                &USER)
        DLTOVR   FILE(*ALL)
ENDPGM

```

Figure 20. T1520CL1 — CL Source to Pass Variables to an ILE C Program

This program passes by reference, the CL variables item name, price, quantity, and user ID to an ILE C program T1520PG1. Variables in a CL program are passed by reference, allowing an ILE C program to change the contents in the CL program.

The Retrieve Job Attributes (RTVJOBA) command obtains the user ID for the audit trail.

Note: The variable *item_name* is null-terminated in the CL program T1520CL1.

Passing CL variables from CL to ILE C are not automatically null-terminated on a compiled CL call. See Chapter 16, “Calling Conventions” on page 325 for information about null-terminated strings for compiled CL calls and command line CL calls.

3. To create CL command prompt T1520CM1 using the source shown below, type:

```
CRTCMD CMD(MYLIB/T1520CM1) PGM(MYLIB/T1520CL1) SRCFILE(QCPPL/QACMDSRC)
```

```

CMD      PROMPT('CALCULATE TOTAL COST')
        PARM      KWD(ITEM) TYPE(*CHAR) LEN(20) RSTD(*NO) +
                MIN(1) ALWUNPRT(*NO) PROMPT('Item name' 1)
        PARM      KWD(PRICE) TYPE(*DEC) LEN(10 2) RSTD(*NO) +
                RANGE(0.01 99999999.99) MIN(1) +
                ALWUNPRT(*YES) PROMPT('Unit price' 2)
        PARM      KWD(QUANTITY) TYPE(*INT2) RSTD(*NO) RANGE(1 +
                9999) MIN(1) ALWUNPRT(*YES) +
                PROMPT('Number of items' 3)

```

Figure 21. T1520CM1 — CL Command Source to Receive Input Data

You use this CL command prompt to enter the item name, price, and quantity for the ILE C program T1520PG1.

4. To compile module T1520IC1 using the source shown below, type:

```
CRTCMOD MODULE(MYLIB/T1520IC1) SRCFILE(QCPPL/QACSRC) OUTPUT(*PRINT) DBGVIEW(*ALL)
```

```

/* This program demonstrates how to use multiple modules, service */
/* programs and a binding directory. This program accepts a user ID, */
/* item name, quantity, and price, calculates the total cost, and */
/* writes an audit trail of the transaction. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <decimal.h>
int calc_and_format (decimal(10,2),
                    short int,
                    char[]);
void write_audit_trail (char[],
                      char[],
                      decimal(10,2),
                      short int,
                      char[]);
int main(int argc, char *argv[])
{
/* Incoming arguments from a CL program have been verified by */
/* the *CMD and null ended within the CL program. */
char *user_id;
char *item_name;
short int *quantity;
decimal (10,2) *price;
char formatted_cost[22];
/*Incoming arguments are all pointers. */
item_name = argv[1];
price = (decimal (10, 2) *) argv[2];
quantity = (short *) argv[3];
user_id = argv[4];
/* Call an ILE C function that returns a formatted cost. */
/* Function calc_and_format returns true if successful. */
if (calc_and_format (*price, *quantity, formatted_cost))
{
write_audit_trail (user_id,
                  item_name,
                  *price,
                  *quantity,
                  formatted_cost);
printf("\n%d %s plus tax = %-s\n", *quantity,
                  item_name,
                  formatted_cost);
}
else
{
printf("Calculation failed\n");
}
return 0;
}

```

Figure 22. ILE C Source to Call Functions in Other Modules

OUTPUT(*PRINT) specifies that you want a compiler listing. The parameter DBGVIEW(*ALL) specifies that you want a root source view and a listing view, along with debug data, to debug this module. See Chapter 7, “Debugging a Program” on page 87 for information on debug views and debug data.

This module has a PEP that is generated by the ILE C compiler during compilation. The PEP is an entry point for the ILE C/C++ program on a dynamic program call from the CL program T1520CL1. The PEP is shown in the call stack as `_C_pep`.

The `main()` function in this module is the UEP which is the target of a dynamic program call from the CL program T1520CL1. The UEP receives control from the PEP.

The `main()` function in this module receives the incoming arguments from the CL program T1520CL1 that are verified by the CL command prompt T1520CM1. All the incoming arguments are pointers. The variable `item_name` is null ended within the CL program T1520CL1.

The `main()` function in this module calls `calc_and_format` in module T1520IC2 to return a formatted cost. If the `calc_and_format` returns successful a record is written to the audit trail by `write_audit_trail` in the service program T1520SP1. The function `write_audit_trail` is not defined in this module (T1520IC1), so it must be imported.

5. To compile module T1520IC2 using the source shown below, type:
`CRTCMOD MODULE(MYLIB/T1520IC2) SRCFILE(QCPPL/QACSRC) OUTPUT(*PRINT)
DBGVIEW(*ALL)`

```

/* This function calculates the tax and formats the total cost. The */
/* function calc_and_format() returns 1 if successful and 0 if it */
/* fails. */
#include <stdio.h>
#include <string.h>
#include <decimal.h>
/* Tax rate is imported from the service program T1520SP2. */
const extern decimal (2,2) taxrate;
int calc_and_format (decimal (10,2) price,
                    short int quantity,
                    char formatted_cost[22])
{
    decimal (17,4) hold_result;
    char hold_formatted_cost[22];
    int i,j;
    memset(formatted_cost, ' ', 21);
    hold_result = (decimal(4,0))quantity *
                  price *
                  (1.00D+taxrate); /* Calculate the total cost. */
    if (hold_result < 0.01D || hold_result > 1989800999801.02D)
    {
        printf("calc out of range:%17.4D(17,4)\n", hold_result);
        return(0);
    }
    /* Format the total cost. */
    sprintf(hold_formatted_cost, "%21.2D(17,4)", hold_result);
    j = 0;

```

Figure 23. ILE C Source to Calculate Tax and Format Cost (Part 1 of 2)


```

for (i=0; i<22; ++i)
{
    if (hold_formatted_cost[i] != ' ' &
        hold_formatted_cost[i] != '0')
    {
        hold_formatted_cost[j] = '$';
        break;
    }
    j = i;
}
for (i=j=21; i>=0; --i)
{
    if (j<0) return(0);
    if (hold_formatted_cost[i] == '$')
    {
        formatted_cost[j] = hold_formatted_cost[i];
        break;
    }
    if (i<16 & !((i-2)%3))
    {
        formatted_cost[j] = ',';
        --j;
    }
    formatted_cost[j] = hold_formatted_cost[i];
    --j;
}
/* End of for loop, 21->0. */
return(1);
}

```

Figure 23. ILE C Source to Calculate Tax and Format Cost (Part 2 of 2)

DBGVIEW(*ALL) specifies that you want a root source view and a listing view, along with debug data to debug this module.

The function `calc_and_format` in this module calculates and formats the total cost. To do the calculation, the data item `taxrate` is imported from service program `T1520SP2`. This data item must be imported because it is not defined in this module (`T1520IC2`).

6. To compile module `T1520IC3` using the following source, type:

```

CRTCMOD MODULE(MYLIB/T1520IC3) SRCFILE(QCPPL/QACSRC)
OUTPUT(*PRINT) DBGVIEW(*SOURCE) OPTION(*SHOWUSR)

```

```

/* This function writes an audit trail. To write the audit trail the */
/* file field structure is retrieved from the DDS file T1520DD1 and */
/* the taxrate data item is imported from service program T1520SP2. */
/* Retrieves the file field structure. */
#pragma mapinc("myinc", "MYLIB/T1520DD1(*all)", "both", "p z","")
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <decimal.h>
#include <recio.h>
#include <xxcvt.h>

/* These includes are for the call to QWCCVTD1 API to get the system */
/* date to be used in the audit trail. */
#include <QSYSINC/H/QWCCVTD1>
#include <QSYSINC/H/QUSEC>
/* DDS mapping of the audit file, T1520DD1. */
#include "myinc"
/* Tax rate is imported from service program T1520SP2. */
const extern decimal (2,2) taxrate;
void write_audit_trail (char          user_id[10],
                        char          item_name[],
                        decimal (10,2) price,
                        short int     quantity,
                        char          formatted_cost[22])
{
    char    char_item_name[21];
    char    char_price[11];
    char    temp_char_price[11];
    char    char_quantity[4];
    char    char_date[6];
    char    char_taxrate[2];
/* Qus_EC_t is defined in QUSEC. */
    Qus_EC_t errcode;
    char    get_date[16];
    int     i;
    double  d;
/* File field structure is generated by the #pragma mapinc directive. */
    MYLIB_T1520DD1_T1520DD1R_both_t buf1;
    _RFILE *fp;
/* Get the current date. */
    errcode.Bytes_Provided = 0;
    QWCCVTD1 ("*CURRENT ", "", "*MDY ", get_date, &errcode);
    memcpy (char_date, &(get_date[1]), 6);

```

Figure 24. ILE C Source to Write an Audit Trail (Part 1 of 2)

```

/*Loop through the item_name and remove the null terminator.      */
for (i=0; i<=20; i++)
{
    if (item_name[i] == '\0') char_item_name[i] = ' ';
    else char_item_name[i] = item_name[i];
}
/*Convert packed to zoned for audit file.                          */
d = price;
QXXDTOZ (char_price, 10, 2, d);
QXXITOZ ( char_quantity, 4, 0, quantity);
d = taxrate;
QXXDTOZ (char_taxrate, 2, 2, d);
/* Set up buffer for write.                                        */
memset(&buf1, ' ', sizeof(buf1));
memcpy(buf1.USER,  user_id, 10);
memcpy(buf1.ITEM,  char_item_name, 20);
memcpy(buf1.PRICE, char_price, 11);
memcpy(buf1.QTY,   char_quantity, 4);
memcpy(buf1.TXRATE, char_taxrate, 2);
memcpy(buf1.TOTAL, formatted_cost, 21);
memcpy(buf1.DATE,  char_date, 6);
if ((fp = _Ropen("MYLIB/T1520DD1", "ar+") == NULL)
{
    printf("Cannot open audit file\n");
}
_Rwrite(fp, (void *)&buf1, sizeof(buf1));
_Rclose(fp);
}

```

Figure 24. ILE C Source to Write an Audit Trail (Part 2 of 2)

The function `write_audit_trail` in this module writes the audit trail for the ILE C program T1520PG1. To write the audit trail, the tax rate is imported from the service program T1520SP2. This module (T1520IC3) is used to create service program T1520SP1.

Note: This source requires 2 members, QUSEC and QWCCVTDT, that are in the QSYSINC/H file. The QSYSINC library is automatically searched for system include files as long as `OPTION(*STDINC)` (the default) is specified on CRTBNDC or CRTCMOD.

`OPTION(*SHOWUSR)` expands the typedefs generated by the compiler from the DDS source file MYLIB/T1520DD1, as specified on the `#pragma mapinc` directive, into the compiler listing and the include debug view. (The include name `myinc` is associated with the temporary source member created by the compiler when it generates the typedefs for the `#pragma mapinc` directive.) See Chapter 10, “Using Externally Described Files in Your Programs” on page 185 for information on how to use the `#pragma mapinc` directive. `DBGVIEW(*SOURCE) OPTION(*SHOWUSR)` specifies that you want an include view containing the root source member, user include files, and debug data to debug this module.

7. Type:

```

CRTCMOD MODULE(MYLIB/T1520IC4) SRCFILE(QCPPL/QACSRC) OUTPUT(*PRINT)
OPTION(*XREF) DBGVIEW(*SOURCE)

```

To compile module T1520IC4 using the following source:

```

/* Export the tax rate data.                                     */
#include <decimal.h>
const decimal (2,2)  taxrate = .15D;

```

Figure 25. T1520IC4 — ILE C Source to Export Tax Rate Data

This service program exports the tax rate data so that it can be used by `calc_and_format` and `write_audit_trail`. The data item `taxrate` is an export because it is coded in this service program so that it can be imported by `calc_and_format` in service program T1520IC2 and the `write_audit_trail` in T1520IC3. This module is used to create service program T1520SP2.

`OPTION(*XREF)` generates a cross reference table containing the list of identifiers in the source code with the numbers of the lines in which they appear. The table provides the class, length and type of the variable `taxrate`. The class is an external definition. The length is 2. The type is a constant decimal(2,2). The use of this option in this example is for illustrative purposes. Typically you use this option when there are several variable references or executable statements. `DBGVIEW(*SOURCE)` creates a root source view and debug data to debug this module.

Note: If you do not specify `DBGVIEW(*SOURCE)` you can debug the modules that reference `taxrate`, but you cannot display `taxrate` during that debug session nor can you debug this module that defines `taxrate`.

8. To create a binding directory entry T1520BD1, type:

```
CRTBNDDIR BNDDIR(MYLIB/T1520BD1)
```

This binding directory is used to contain the names of two service programs T1520SP1 and T1520SP2 that are needed to create service program T1520SP1 and ILE C program T1520PG1.

9. To add two service program names to the binding directory T1520BD1, type:

```
ADDBNDDIRE BNDDIR(MYLIB/T1520BD1) OBJ((MYLIB/T1520SP1 *SRVPGM))
ADDBNDDIRE BNDDIR(MYLIB/T1520BD1) OBJ((MYLIB/T1520SP2 *SRVPGM))
```

The service program names T1520SP1 and T1520SP2 can be added even though the service program objects do not exist yet.

10. Type:

```
CRTSRCPF FILE(MYLIB/QSRVSRC) MBR(T1520SP2)
```

11. To create a source physical file QSRVSRC using the following binder language source:

```
STRPGMEXP PGMLVL(*CURRENT)
          EXPORT    SYMBOL('taxrate')
ENDPGMEXP
```

Figure 26. Binder Language Source to Export Tax Rate Data

Use the binder language to define the export data item `taxrate` of the service program T1520SP2. See the binder language in QCPPL/QRVSRC member T1520SP2. BND is the source type for binder language source.

A tool is provided in the QUSRTOOL library to help generate the binder language for one or more modules. See member TBNINFO in the file

QUSRTOOL/QATTINFO. The **binder language** allows you to define the data item name `taxrate` that can be exported.

The Start Program Export (STRPGMEXP) command identifies the beginning of a list of exports from the service program T1520SP2. The Export Symbol (EXPORT) command identifies the symbol name `taxrate` to be exported from the service program T1520SP2.

Note: The symbol name `taxrate` is enclosed in apostrophes to maintain its lowercase format. Without apostrophes it is converted to uppercase characters. (The binder would search for `TAXRATE` which it would not find.)

The End Program Export (ENDPGMEXP) command identifies the end of the exports from the service program T1520SP2.

The symbol name `taxrate` identified between the STRPGMEXP PGMLVL(*CURRENT) and ENDPGMEXP pair defines the public interface to the service program T1520SP2. Only procedure names and data item names exported from the module objects making up the ILE C service program can be exported from this service program. This is the public interface.

The symbol name `taxrate` is also used to create a signature. The signature validates the public interface to the service program T1520SP2 at activation. This ensures that the ILE C service program T1520SP1 and the ILE C program T1520PG1 can use service program T1520SP2 without being re-created.

The default for the LVLCHK parameter on the STRPGMEXP command is *YES. This causes the binder to generate a nonzero signature value. If you specify LVLCHK(*NO) the binder generates a signature value of zeros. Use LVLCHK(*NO) with caution because it means that you are responsible for manually verifying that the public interface is compatible with previous levels. Specify LVLCHK(*NO) only if you can control which procedures of the ILE C service program are called and which variables are used by other ILE C objects. If you cannot control the public interface, run-time or activation errors may occur.

12. To create service program T1520SP2 from module T1520IC4 and the binder source language in T1520SP2, type:

```
CRTSRVPGM SRVPGM(MYLIB/T1520SP2) MODULE(MYLIB/T1520IC4)
SRCFILE(QCPPLE/QASRVSRC) SRCMBR(*SRVPGM) DETAIL(*FULL)
```

Member T1520SP2 is in the source file QASRVSRC. Member T1520SP2 contains the binder language source that translates into a signature and an export identifier. This service program (T1520SP2) exports a data item `taxrate` to satisfy an import request from `calc_and_format` in module T1520IC2.

Note: No binding directory is needed because the module needed to create the service program is specified on the module parameter.

The default EXPORT(*SRCFILE) is used. The binder is directed to the SRCFILE and SRCMBR parameters to locate binder language source and generate the signature from the order of modules processed and of the symbols exported.

If EXPORT(*ALL) is specified, no binder language source is needed to define the exports for the service program. An ILE C service program with

EXPORT(*ALL) is difficult to update once the exports are used by other ILE C programs. If the service program is changed, the order or number of exports could change. If the signature changes all ILE C programs and service programs that use the changed service program have to be re-created.

ACTGRP(*CALLER) is the default activation group for the CRTSRVPGM command. It specifies that when service program T1520SP2 is called it is activated into the caller's activation group, the ILE C program T1520PG1 activation group.

This service program is created with the default OPTION(*RSLVREF). This option ensures that the program object T1520SP2 is created only if all external references to bind multiple objects into a service program object are resolved. When you are developing a program you may want to create an ILE C program or service program with unresolved references. See "Handling Unresolved Import Requests with *UNRSLVREF" on page 47.

13. Type:

```
CRTSRCPF FILE(MYLIB/QSRVSRC) MBR(T1520SP1)
```

14. To create a source physical file QSRVSRC using the following binder language source:

```
STRPGMEXP PGMLVL(*CURRENT)
          EXPORT      SYMBOL('write_audit_trail')
          ENDPGMEXP
```

Figure 27. Binder Language source to Export write_audit_trail Procedure

Use the binder language to define the export procedure name write_audit_trail of the service program T1520SP1. See the binder language in QCPPL/ QASRVSR member T1520SP1. BND is the source type for the binder language source.

15. To create service program T1520SP1 from module T1520IC3, type:

```
CRTSRVPGM SRVPGM(MYLIB/T1520SP1) MODULE(MYLIB/T1520IC3) SRCFILE(QCPPL/ QASRVSR)
SRCMBR(*SRVPGM) BNDDIR(MYLIB/T1520BD1) DETAIL(*FULL)
```

A binding directory T1520BD1 is used to find the service program (T1520SP2) that contains taxrate. Any other program that needs to use the function write_audit_trail or data item taxrate can use this binding directory.

Service program T1520SP1 exports a procedure write_audit_trail to satisfy an import request from function write_audit_trail in module T1520IC1.

16. To create program T1520PG1 from modules T1520IC1 and T1520IC2 using the binding directory T1520BD1, type:

```
CRTPGM PGM(MYLIB/T1520PG1) MODULE(MYLIB/T1520IC1 MYLIB/T1520IC2) ENTMOD(*ONLY)
BNDDIR(MYLIB/T1520BD1) DETAIL(*FULL)
```

Module T1520IC1 contains an import procedure request named write_audit_trail which matches an export request in T1520SP1 using symbol write_audit_trail. The binder matches the import request from T1520IC1 for the procedure write_audit_trail with the corresponding export from T1520SP1 and the import request from T1520IC2 for the data taxrate with the corresponding export from T1520SP2.

BNDDIR(MYLIB/T1520BD1) specifies that the binding directory T1520BD1 in library MYLIB is to be searched to resolve the import requests from modules T1520IC1 and T1520IC2. The binding directory T1520BD1 contains the names of the two service programs T1520SP1 and T1520SP2. References to ILE C run-time functions are resolved because they are automatically bound into any ILE C program containing ILE C modules.

The default OPTION(*RSLVREF) ensures that the program object T1520PG1 is created only if all external references are resolved. When you are developing an ILE C program or service program you may want to create an ILE C program or service program with unresolved references. See “Handling Unresolved Import Requests with *UNRSLVREF” on page 47.

ENTMOD(*ONLY) specifies that only one module from the list of modules can have a PEP. An error is issued if more than one module is found to have a PEP. If ENTMOD(*FIRST) is used then the first module found from a list of modules, that has a PEP is selected as the PEP. All other modules with PEPs are ignored.

The default ACTGRP(*NEW) is used to define a new activation group. The program T1520PG1 is associated with a new activation group. Service programs T1520SP1 and T1520SP2 were created with activation group *CALLER. These service programs become part of the callers activation group using the resources of one activation group for the purposes of developing one cooperative program. Service program T1520SP1 and T1520SP2 are bound to the program being activated. These service programs are also activated as part of the dynamic call processing.

17. To run the program T1520PG1, type T1520CM1 and press F4 (Prompt).

Type the following data into T1520CM:

The output is as shown:

```
Hammers
1.98
5000
Nails
0.25
2000
```

Figure 28. Sample Data for Program T1520PG1

```
5000 HAMMERS plus tax =          $11,385.00
Press ENTER to end terminal session.
>
2000 NAILS plus tax =           $575.00
Press ENTER to end terminal session.
```

The physical file T1520DD1 is updated with the following data:

SMITHE	HAMMERS	00000000198500015	\$11,385.00072893
SMITHE	NAILS	0000000025200015	\$575.00072893

Figure 29. Output File from Program T1520PG1

Chapter 6. Running a Program

This chapter describes how to:

- Run an ILE C/C++ program using the CALL command
- Pass parameters to a program
- Run an ILE C/C++ program using the Transfer Control (TFRCTL) command
- Use an ILE C/C++ program as a Command Processing Program (CPP)

In addition to the CALL command, the TFRCTL command, and as a CPP there are several other ways to run a program. You can use:

- The Programmer Menu..
- The Start Programming Development Manager (STRPDM) command..
- A high-level language CALL statement. Chapter 16, “Calling Conventions” on page 325 contains information on interlanguage calls.
- The EVOKE statement from an ICF file.
- The QCAPCMD program.
- An ILE C/C++ program can be called by the REXX interpreter.

The ILE C/C++ Run-Time Model

The ILE C/C++ run-time model guarantees ANSI C/C++ semantics when all programs in an application are created with the CRTBND and CRTBNDCLP Create Bound Program commands, or with the following options on the CRTPGM command:

ACTGRP(*NEW)

A new activation group is created on every call of the created *PGM, and the activation group is destroyed when the program ends.

OPTION(*NODUPPROC)

No duplicate procedure definitions in the same bound program are allowed.

OPTION(*NODUPVAR)

No duplicate variable definitions in the same bound program are allowed.

Note: When a CRTPGM parameter does not appear in the Create Bound Program command invocation, the default is the CRTPGM parameter. For example, the parameter ACTGRP(*NEW) is the default for the CRTPGM command, and is used for the Create Bound Program command. You can change the CRTPGM parameter defaults using the Change Command Defaults (CHGCMDDFT) command.

The ILE C/C++ run-time library functions are bound to the application in the same activation group in which it is called. Therefore all program activations in the same activation group share one instance of the ILE C/C++ run-time library and the state of the ILE C/C++ run time propagates across program call boundaries. That is, if one program in an activation group changes the state of the ILE C/C++ run time, then all other programs in that activation group are affected. For example, other programs in the same activation group are affected by the locale setting of an application or the multibyte functions' shift-in/shift-out state.

If the ACTGRP parameter of the CRTPGM command is specified to a value other than *NEW, the application's run-time behavior may not follow ANSI C semantics. Non-ANSI behavior may occur during:

- Program ending (exit(), abort(), atexit())
- Signal handling (signal(), raise())
- Multibyte string handling (mblen())
- Any locale dependent library functions (isalpha(), qsort()).

In the default activation groups, I/O files are not automatically closed. The I/O buffers are not flushed.

If ACTGRP is set to *CALLER, multiple calls of an ILE C/C++ program share one instance of the ILE C/C++ run-time library state in the same activation group. Through this option, ILE C/C++ programs can run within the OPM default activation groups. Certain restrictions exist for ILE C/C++ programs that run in the OPM default activation groups. For example, you are not allowed to register atexit() functions within the OPM default activation groups.

If the activation group is named, all calls to programs in this activation group within the same job share the same instance of the ILE C/C++ run-time library state.

It is possible to create an ANSI compliant application whose programs are created with options other than ACTGRP(*NEW). However, it is the responsibility of the application designer to ensure that the sharing of resources, and run-time states across all programs in the activation group do not result in non-ANSI behavior.

Activation Groups

After successfully creating an ILE C/C++ program, you can run your code. Activation is the process of getting an ILE C/C++ program or service program ready to run. When an ILE C/C++ program is called, the system performs activation. Because ILE C/C++ service programs are not called, they are activated during the call to an ILE C/C++ program that directly or indirectly requires their services.

Activations and activation groups:

- Help ensure that ILE C/C++ programs running in the same job run independently without intruding on each other by scoping resources to the activation group. Example of programs running in the same job run are commitment control, overrides, and shared files.
- Scope resources to the ILE C/C++ program.
- Uniquely allocate static data needed by the ILE C/C++ program or service program.
- Change symbolic links to ILE C/C++ service programs to physical addresses.

When activation allocates the storage necessary for the static variables that are used by an ILE C/C++ program, the space is allocated from an activation group. At the time the ILE C/C++ program or service program is created, you specify the activation group that should be used at run time.

Once an ILE C/C++ program is activated, it remains activated until the activation group is deleted. Even though they are activated, programs do not appear in the call stack unless they are running.

When an OS/400 job is started, the system creates two activation groups for OPM programs. One activation group is reserved for OS/400 system code and the other is used for all other OPM programs. You cannot delete the OPM default activation groups. The system deletes them when your job ends.

An activation group can continue to exist even when the `main()` function of an ILE C/C++ program is not on the call stack. This occurs when the ILE C/C++ program was created with a named activation group (specifying a name on the `ACTGRP` option of the `CRTPGM` command), and the `main()` function issues a return. This can also occur when the ILE C/C++ program performs a `longjmp()` across a control boundary by using a jump buffer that is set in an ILE C/C++ procedure. This procedure is higher in the call stack and before the nearest control boundary.

Calling a Program

When you call a program, the OS/400 system locates the corresponding executable code and performs the instructions found in the program.

Note: Only programs can run independently. Service programs or other bound procedures must be called from a program that requires their services.

There are several ways to call a program:

- Issue the `CL CALL` command.
- Issue the `CL Transfer Control (TFRCTL)` command.
- Issue a user-created `CL` command.

In addition, you can run a program using:

- The Programmer Menu
- The Start Programming Development Manager (`STRPDM`) command.
- The `EVOKE` statement from an ICF file.
- The `QCAPEXC` program.
- A high-level language. See Chapter 3, “Service Programs” on page 33 for information on calling service programs and procedures).
- The REXX interpreter.

Calling a Program Using the TFRCTL Command

You can run an application from within a `CL` program that transfers control to your program using the Transfer Control (`TFRCTL`) command. This command:

1. Transfers control to the program specified on the command.
2. Removes the transferring `CL` program from the call stack.

In the following example, the `TFRCTL` command in a `CL` program `RUNCP` calls a C++ program `XRUN2`, which is specified on the `TFRCTL` command. `RUNCP` transfers control to `XRUN2`. The transferring program `RUNCP` is removed from the call stack.

Figure 30 illustrates the call to the `CL` program `RUNCP`, and the transfer of control to the C++ program `XRUN2`.

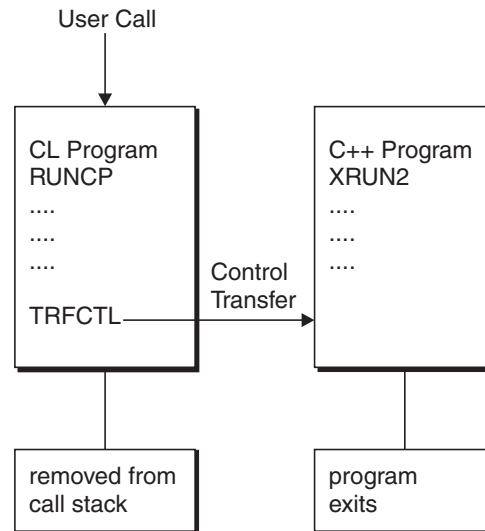


Figure 30. Calling Program XRUN2 Using the TRFCTL Command

To create and run programs RUNCP and XRUN2, follow the steps below:

1. Enter the source code below into a source physical file QCLSRC, in library MYLIB.

```

/* Source for CL Program RUNCP                                     */
PGM          PARM(&STRING)
DCL          VAR(&STRING)  TYPE(*CHAR)  LEN(20)
DCL          VAR(&NULL)    TYPE(*CHAR)  LEN(1)  VALUE(X'00')

/* ADD NULL TERMINATOR FOR THE ILE C++ PROGRAM                     */
CHGVAR       VAR(&STRING)  VALUE(&STRING *TCAT &NULL)
TRFCTL       PGM(MYLIB/XRUN2) PARM(&STRING)

/* THE DSPJOBLOG COMMAND IS NOT CARRIED OUT SINCE                 */
/* WHEN PROGRAM XRRUN2 RETURNS, IT DOES NOT RETURN TO THIS      */
/* CL PROGRAM.                                                    */
DSPJOBLOG
ENDPGM

```

2. Create the CL program RUNCP. From the command line, type:
CRTCLPGM PGM(MYLIB/RUNCP) SRCFILE(MYLIB/QCLSRC)

Program RUNCP uses the TRFCTL command to pass control to the ILE C++ program XRUN2, which does not return to RUNCP.

3. Create program XRUN2 in library MYLIB from source file xrun2.cpp, shown below:

```

// xrun2.cpp
// Source for Program XRUN2
// Receives and prints a null-terminated character string

#include <iostream.h>

int main(int argc, char *argv[])
{
    int    i;
    char * string;
    string = argv[1];
    cout << "string = " << string << endl;
}

```

Program XRUN2 receives a null terminated character string from the CL program and prints the string.

4. Run program RUNCP from a command line, passing it the string "nails", with the command:
`CALL PGM(MYLIB/RUNCP) PARM('nails')`

The output from program XRUN2 is:

```
string = nails
Press ENTER to end terminal session.
```

Running a Program from a User-Created CL Command

You can also run a program from your own CL command. To create a command:

1. Enter a set of command statements into a source file.
2. Process the source file and create a command object (type *CMD) using the Create Command (CRTCMD) command.

The CRTCMD command definition includes the command name, parameter descriptions, and validity-checking information, and identifies the program that performs the function requested by the command.

3. Enter the command interactively, or in a batch job.

The program called by your CL command is run.

The following example illustrates how to run a program from a user-created CL command:

Program Description

A newly created command COST prompts for and accepts user input values. It then calls a C++ program CALCOST and passes it the input values. CALCOST accepts the input values from the command COST, performs calculations on these values, and prints results. Figure 31 illustrates this example.

Command

COST

- Prompts for user input
- Accepts user-input values
- Calls program CALCOST and passes input values to it

C++ Program

CALCOST

- Processes input values
- Performs calculations
- Produces printed output

Figure 31. Calling Program CALCOST from a User-Defined Command COST

To create and run the example, follow the steps below:

1. Enter the source code for the command prompt COST shown below into a source file QCMSRC in library MYLIB:

```
// Source for Command Prompt COST
CMD      PROMPT('CALCULATE TOTAL COST')
PARM     KWD(ITEM) TYPE(*CHAR) LEN(20) RSTD(*NO) +
        MIN(1) ALWUNPRT(*NO) PROMPT('Item name' 1)
PARM     KWD(PRICE) TYPE(*DEC) LEN(10 2) RSTD(*NO) +
        RANGE(0.01 99999999.99) MIN(1) +
        ALWUNPRT(*YES) PROMPT('Unit price' 2)
PARM     KWD(QUANTITY) TYPE(*INT2) RSTD(*NO) RANGE(1 +
        9999) MIN(1) ALWUNPRT(*YES) +
        PROMPT('Number of items' 3)
```

2. Create the CL command prompt COST. On a command line, enter:

```
CRTCMD CMD(MYLIB/COST) PGM(MYLIB/CALCOST) SRCFILE(MYLIB/QCMDSRC)
```

Use the CL command prompt COST to enter item name, price, and quantity for the ILE C++ program CALCOST.

3. Create program CALCOST from the source file calcost.cpp shown below:

```
// calcost.cpp
// Source for Program CALCOST

#include <iostream.h>
#include <string.h>
#include <bcd.h>

int main(int argc, char *argv[])
{
    char            *item_name;
    _DecimalT<10,2> *price;
    short int       *quantity;
    const _DecimalT<2,2> taxrate=__D("0.15");
    _DecimalT<17,2> cost;
    item_name = argv[1];
    price     = (_DecimalT<10,2> *) argv[2];
    quantity  = (short *) argv[3];
    cost = (*quantity)*(*price)*(__D("1.00")+taxrate);
    cout << "\nIt costs $" << cost << " to buy "
        << *quantity << " " << item_name << endl;
}
```

This program receives the incoming arguments from the CL command COST, calculates a cost, and prints values. All incoming arguments are pointers.

4. Enter data for the program CALCOST. From the command line, type COST and press F4 (Prompt).

Type the data shown below into COST:

```
Hammers
1.98
5000
Nails
0.25
2000
```

The output of program CALCOST is:

```
It costs $11385.00 to buy 5000 HAMMERS
Press ENTER to end terminal session.
>
It costs $575.00 to buy 2000 NAILS
Press ENTER to end terminal session.
```

Using the Call Command

You can run a program by using the CALL command. For example, to call program T1520ALP, type:

```
CALL PGM(MYLIB/T1520ALP)
```

The program object must exist. In this example, the library MYLIB is not in the library list *LIBL. Since the program object is not in the library list, the library name is specified on the CALL command.

Using the CL CALL Command

From the command line, you can use the CALL command to run a program interactively, or as part of a batch job.

The syntax for this CL command is:

```
►►—CALL PGM—(library-name/program-name)—►►
```

For example, the command

```
CALL PGM(MYLIB/MYPROG)
```

invokes the program MYPROG located in the library MYLIB.

If the program object specified by program-name exists in a library that is contained in your library list, you can omit the library name in the command, and the syntax is:

```
►►—CALL—program-name—►►
```

For example, if MYLIB appears in your library list, you can simply type:

```
CALL MYPROG
```

If you need prompting for the command parameters, type CALL and press F4 (Prompt). If you need help, type CALL and press F1 (Help).

Passing Parameters to a Program

When you request prompting with the CALL command, a display appears that allows you to supply the parameters to the program you are calling. You can also type the parameters directly following the CALL command.

Example

The following example shows an ILE C/C++ program T1520REP that requires parameters at run time.

1. To create the program T1520REP using the source shown below, type:

```
CRTBNDC PGM(MYLIB/T1520REP) SRCFILE(QCPPL/QACSRC)
```

```

/* Print out the command line arguments.                                */
#include <stdio.h>
void main ( int argc, char *argv[] )
{
    int i;
    for ( i = 1; i < argc; ++i )
        printf( "%s\n", argv[i] );
}

```

Figure 32. T1520REP — ILE C Source to Pass Parameters to an ILE C Program

2. To run the program T1520REP, type:

```
CALL PGM(MYLIB/T1520REP) PARM('Hello, World')
```

The output is as shown:

```

Hello, World
Press ENTER to end terminal session.

```

When you call a program from a CL command line, the parameters you pass on the CALL command are changed as follows:

- String literals are passed with a null terminating character.
- Numeric constants are passed as packed decimal digits.
- Characters that are not enclosed in single quotation marks are folded to uppercase, and are passed with a null character.
- Characters that are enclosed in single quotation marks are not changed, and mixed case strings are supported, and are passed with a null terminating character.

For example:

```

CALL PGM(T1520REP) PARM(abc)      - ABC\0 (converted to uppercase; passed as a string)
CALL PGM(T1520REP) PARM('123.4') - 123.4\0 (passed as a string)
CALL PGM(T1520REP) PARM(123.4)    - 123.4 (passed as a packed decimal (15,5))
CALL PGM(T1520REP) PARM('abc')    - abc\0 (passed as a string)
CALL PGM(T1520REP) PARM('abC')    - abC\0 (passed as a string)

```

You can use the QCAPCMD program to add the null character to arguments that are passed to an ILE C/C++ program.

Note: The Process Commands (QCAPCMD) API is used to perform command analyzer processing on command strings. You can check or run CL commands from HLLs as well as check syntax for specific source definition types. You can use the QCAPCMD API to check the syntax of a command string prior to running it, prompt the command and receive the changed command string, and run a command from an HLL.

The REXX interpreter treats all REXX variables as strings (without a null terminator). REXX passes parameters to OS/400 which then calls the ILE C/C++ program. Conversion to a packed decimal data type still occurs, and strings are null terminated.

Note: These changes only apply to calling a program from a command line, not to interlanguage calls. See Chapter 16, “Calling Conventions” on page 325 for information on ILE C/C++ calling conventions.

To pass parameters to an ILE program when you run it, use the *PARM* option of the CL CALL command. The syntax for this command is:

```
CALL PGM(program-name) PARM(param-1 param-2 ... param-n)
```

You can also type the parameters without specifying any keywords:

```
CALL library/program-name (param-1 param-2 ... param-n)
```

The following example demonstrates how to pass the value 'Hello, World' to program XRUN1 which expects parameters at run time. The source file xrun1.cpp for program XRUN1 is shown below:

```
// xrun1.cpp
// Prints out command line arguments.
```

```
#include <iostream.h>
int main ( int argc, char *argv[])
{
    int i;
    for ( i = 1; i < argc; ++i )
        cout << argv[i] << endl;
}
```

Follow the steps below, to create and run program XRUN1:

1. Compile the source shown above with default compiler options. On the command line type:

```
CRTBNDCPP MODULE(MYLIB/XRUN1) SRCSTMF('xrun1.cpp')
```

The resulting module and program objects are created into the default library, in this example, MYLIB.

2. Run the program from a command line using the command:

```
CALL PGM(MYLIB/XRUN1) PARM('Hello, World')
```

The output of program XRUN1 is:

```
Hello, World
Press ENTER to end terminal session.
```

Ending a Program

When a program ends normally, the system returns control to the caller. The caller could be a workstation user or another program.

If a program ends abnormally during run time, and the program had been running in a different activation group from its caller, the escape message CEE9901 is issued and control is returned to the caller:

```
Application error <msgid> unmonitored by <pgm> at
statement <stmtid>, instruction <instruction>
```

A CL program can monitor for this exception by using the Monitor Message (MONMSG) command.

If the program is running in the same activation group as is its caller and the program ends abnormally, what message is issued depends on how the program ended. If it ended with a function check, CPF9999 is issued.

Managing Activation Groups

Activation groups make it possible for multiple ILE programs to run in the same job independently, without intruding on each other.

An activation group is a substructure of a job. It consists of system resources such as storage, commitment definitions, and open files. These resources are allocated to run one or more ILE or OPM programs. For example, the storage space for the static variables of a program is allocated from an activation group.

Once a program (type *PGM) is called, it remains activated until the activation group it runs in is deleted. Because service programs are not called directly, they are activated during the call to the program that requires their services.

Specifying an Activation Group

When an OS/400 job is started, the system automatically creates two activation groups to be used by OPM programs. One activation group is reserved for OS/400 system code. The other activation group is used for all other OPM programs. The symbol used to represent this activation group is *DFTACTGRP. You cannot delete the OPM default activation groups. The system deletes them when your job ends.

Note: OPM programs always run in the default activation group; you cannot change their activation group specification.

For ILE programs you specify the activation group that should be used at run-time through the ACTGRP parameter of the Create Program or Create Service Program commands. You can choose between:

- Running your program in a named activation group.
- Accepting the default activation group:
 - *NEW for programs
 - *CALLER for service programs.
- Activating a program into the activation group of a calling program.

Running a Program in a Named Activation Group

To manage a collection of ILE programs and service programs as one application, you create a named activation group for them by specifying a user-defined name on the ACTGRP parameter.

The system creates the named activation group as soon as the first program that has specified this activation group is called. This group is then used by all programs and service programs that have specified its name.

A named activation group ends when it is deleted through the Reclaim Activation Group (RCLACTGRP) command. This command can only be used when the activation group is no longer in use. It also ends when you call the `exit()` function in your code.

When a named activation group ends, all resources associated with the programs and service programs of the group are returned to the system.

Note: Using named activation groups may result in non-ANSI compliant run-time behavior. If a program created using named activation groups remains activated by a return statement, you encounter the following problems:

- Static variables are not reinitialized.

- Static constructors are not called again.
- Static destructors are not called on return.
- Other programs activated in the same activation group may terminate your program, although they seem to be independent.
- Your program is not portable, if you count on the behavior of the named activation group.

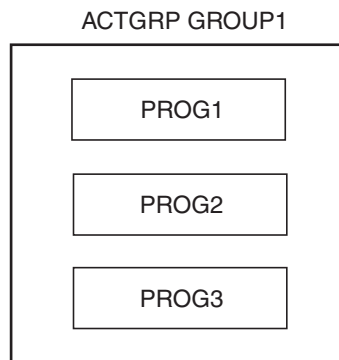


Figure 33. Running Programs in a Named Activation Group

In the following example, programs PROG1, PROG2, and PROG3 are part of the same application and run in the same activation group, GROUP1. Figure 33 illustrates this scenario:

To create these programs in the same activation group, you specify GROUP1 on the *ACTGRP* parameter when you create each program:

```

crtpgm pgm(prog1) actgrp(group1) prog1.cpp
crtpgm pgm(prog2) actgrp(group1) prog2.cpp
crtpgm pgm(prog3) actgrp(group1) prog3.cpp
  
```

Running a Program in Activation Group *NEW

To create a new activation group whenever your program is called, specify **NEW* on the *ACTGRP* parameter. In this case, the system creates a name for the activation group that is unique within your job.

**NEW* is the default value of the *ACTGRP* parameter on the *CRTPGM* command. An activation group created with **NEW* always ends when the last program associated with it ends.

Note: **NEW* is not valid for a service program, which can only run in the activation group of its caller, or in a named activation group.

If you create a program with *ACTGRP(*NEW)*, more than one user can call the program at the same time without using the same activation group. Each call uses a new copy of the program. Each new copy has its own data and opens its files.

In the following example, programs PROG4, PROG5, and PROG6 run in separate unnamed activation groups.

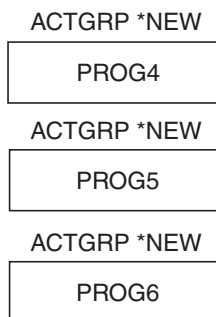


Figure 34. Running Programs in Unnamed Activation Groups

By default, each program is created into a different activation group, identified by the *ACTGRP* parameter (**NEW*).

```

crtpgm pgm(prog4) actgrp(*NEW) prog4.cpp
crtpgm pgm(prog5) actgrp(*NEW) prog5.cpp
crtpgm pgm(prog6) actgrp(*NEW) prog6.cpp

```

Because **NEW* is the default, you obtain the same result with the following invocations:

```

crtpgm pgm(prog4) prog4.cpp
crtpgm pgm(prog5) prog5.cpp
crtpgm pgm(prog6) prog6.cpp

```

Note: If you invoke three source files in one command, a single program object *PROG* is created in activation group **NEW*:

```
crtpgm pgm(prog) prog7.cpp prog8.cpp prog9.cpp
```

Non-ANSI Behavior with Named Activation Groups

If the *ACTGRP* parameter of the *CRTPGM* command is specified as a value other than **NEW*, the application's run-time behavior may not follow ANSI semantics. Run-time and class libraries assume that programs are built with *ACTGRP(*NEW)*.

Non-ANSI behavior may occur during:

- Program termination - *exit()*, *abort()*, *atexit()*
- Signal handling - *signal()*, *raise()*
- Multibyte string handling - *mblen()*
- Any locale-dependent library functions - *isalpha()*, *qsort()*.

In the default activation groups, I/O files are not automatically closed. The I/O buffers are not flushed unless explicitly requested.

Running a Program in Activation Group (**CALLER*)

You can specify that an ILE program or an ILE service program be activated within the activation group of a calling program, by setting *ACTGRP* to **CALLER*. With this attribute, a new activation group is never created when the program or service program is activated. Through this option, ILE C/C++ programs can run within the OPM default activation groups when the caller is an OPM program.

Certain restrictions exist for ILE C/C++ programs running in the OPM default activation groups. For example, you are not allowed to register *atexit()* functions within the OPM default activation groups.

If the activation group is named, all calls to programs in this activation group within the same job share the same instance of the ILE C/C++ run-time library state.

It is possible to create an ANSI-compliant application whose programs are created with options other than *ACTGRP(*NEW)*. While non-ANSI behavior may be desirable in certain cases, it is the responsibility of the application designer to ensure that the sharing of resources and run-time states across all programs in the activation group does not result in incorrect behavior.

In the following example, a service program SRV1 is activated into the respective activation groups of programs PROG7 and PROG8. PROG7 runs in a named activation group GROUP2, while PROG8 runs in an unnamed activation group **NEW*.

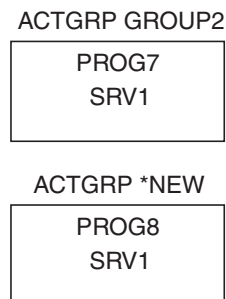


Figure 35. Running a Service Program in the Activation Groups of Calling Programs

By default, the service program SRV1 is created into the activation group of each calling program.

```
crtsrvpgm srvgm(srv1) srv1.cpp
```

Presence of a Program on the Call Stack

Even though it is activated, a program does not appear on the call stack unless it is running. But an activation group can continue to exist even when the *main()* function of the program is not on the call stack.

This occurs when the program was created with a named activation group, and the *main()* function issues a return. It can also occur when the program performs a *longjmp()* across a control boundary by using a jump buffer that is set in an ILE C or C++ procedure. (This procedure is higher in the call stack and before the nearest control boundary.)

Deleting an Activation Group

When an activation group is deleted, its resources are reclaimed. The resources include static storage and open files. A **NEW* activation group is deleted when the program it is associated with returns to its caller.

Named activation groups are persistent. You must delete them explicitly. Otherwise they end only when the job ends. The storage associated with programs running in named activation groups is not released until these activation groups are deleted.

The OPM default activation group is also a persistent activation group. The storage associated with ILE programs running in the default activation group is released either when you sign off (for an interactive job) or when the job ends (for a batch job).

Reclaiming System Resources

You may encounter situations where system storage is exhausted, for example:

- If many ILE programs are activated (that is, called at least once).
- If ILE programs that use large amounts of static storage run in the OPM default activation group (storage is not reclaimed until the job ends).
- If many service programs are called into named activation groups (resources are only reclaimed when the job ends).

In such situations, you may want to reclaim system resources that are no longer needed for a program, but are still tied up because an activation group has not been deleted. You have the following options:

- Delete a named activation group that is not in use through the Reclaim Activation Group (RCLACTGRP) command .

The command provides options to either delete all eligible activation groups or to delete an activation group by name.

- Free resources for programs that are no longer active through the Reclaim Resources (RCLRSC) command.

Using the Reclaim Resources Command (RCLRSC)

The RCLRSC command works differently depending on how the program was created:

- For OPM programs, the RCLRSC command closes open files and frees static storage.
- For ILE programs that were activated into the OPM default activation group (because they were created with *CALLER), The RCLRSC command closes files and reinitializes storage. However, the storage is not released.
- For ILE programs associated with a named activation group, the RCLRSC command has no effect. You must use the RCLACTGRP command to free resources in a named activation group.

Managing Run-Time Storage

ILE allows you to directly manage run-time storage from your program, by managing heaps. A heap is an area of storage used for allocations of dynamic storage. The amount of dynamic storage required by an application depends on the data being processed by the programs and procedures that use the heap. You manage heaps by using ILE bindable APIs.

You are not required to explicitly manage run-time storage. However, you may wish to do so if you want to make use of dynamically allocated storage, for example, if you do not know exactly how big an array should be. In this case you could acquire the actual storage for the array at run-time, once your program determines how big the array should be.

There are two types of heaps available on the system:

- The default heap
- A user-created heap.

You can use one or more user-created heaps to isolate the dynamic storage required by some programs and procedures within an activation group.

The rest of this section explains how to use a default heap to manage run-time storage in a program.

Managing the Default Heap

The first request for dynamic storage within an activation group results in the creation of a **default heap** from which the storage allocation takes place. Additional requests for dynamic storage are met by further allocations from the default heap. If there is insufficient storage in the heap to satisfy the current request for dynamic storage, the heap is extended, and the additional storage is allocated.

Allocated dynamic storage remains allocated until it is explicitly freed, or until the heap is discarded. The default heap is discarded only when the owning activation group ends.

Programs in the same activation group all use the same default heap. If one program accesses storage beyond what has been allocated, it can cause problems for another program.

For example, assume that two programs, PGM1 and PGM2 are running in the same activation group. 10 bytes are allocated for PGM1, but 11 bytes are changed by it. If the extra byte was in fact allocated for PGM2 problems may arise for PGM2.

Using Bindable APIs to Manage the Default Heap

You can use the following ILE bindable APIs on the default heap:

Free Storage (CEEFRST)

Frees one previous allocation of heap storage

Get Heap Storage (CEEGTST)

Allocates storage within a heap

Reallocate Storage (CEEZST)

Changes the size of previously allocated storage.

Dynamically Allocating Storage at Run Time



In an ILE C++ program, you manage dynamic storage belonging to the default heap using the operators `new` and `delete` to create and delete dynamic objects. Dynamic objects are never created and deleted automatically. Their creation can fail if there is not enough free heap space available, and your programs must provide for this possibility.

The following examples illustrate dynamic storage allocation with `new` and `delete`:

1. Dynamic allocation and de-allocation of storage for a class object:

```
TClass *p;                // Define pointer
p= new TClass;            // Construct object
if (!p) {
    Error("Unable to construct object");
    exit(1);
}
...
delete p;                // Delete object
```

2. Dynamic allocation and de-allocation of storage for an array of objects:

```
TClass *array             // Define pointer
array = new TClass[100]; // Construct array of 100 objects
...
delete[] array;           // Delete array
```

Note: In this example, you use `delete[]` to delete the array. Without the brackets, `delete` deletes the entire array, but calls the destructor only for the first element in the array. If you have an array of values that do not have destructors, you can use `delete` or `delete[]`.

Messaging Support

Severity	Compiler Response
Informational (00)	Compilation continues. The message reports conditions found during compilation.
Warning (10)	Compilation continues. The message reports valid, but possibly unintended, conditions.
Error (20)	Compilation continues and object code is generated. Error conditions exist that the compiler can correct, but the program might not run correctly.
Severe error (30)	Compilation continues, but object code is not generated. Error conditions exist that the compiler cannot correct.
Unrecoverable error (40)	The compiler halts. An internal compiler error has been found. This message should be reported to your IBM service representative.

Part 3. Debugging Programs

This part describes:

- How to prepare and compile your program to include debugging data
- How to start and proceed through a debugging session
- How to set breakpoints and watch conditions
- Debugging language syntax

Chapter 7. Debugging a Program

Debugging allows you to detect, diagnose, and eliminate run-time errors in a program. You can debug ILE and OPM programs using the ILE source debugger.

This chapter describes how to:

- Start a debug session
- Add and remove programs from a debug session
- View the program source from a debug session
- Set and remove breakpoints and watch conditions
- Step through the program
- Display and change the value of variables, expressions, structures, or arrays
- Equate a shorthand name with a variable, expression, or command
- Change module optimization and observability

The ILE source debugger is used to help you locate programming errors in ILE C/C++ programs and service programs.

Before you can use the ILE source debugger, you must use one of the non-default debug options (DBGVIEW) when you compile a source file. Next, you can start your debug session. Once you set breakpoints or other ILE source debugger options, you can call the program.

Avoiding Modification of Production Files

While debugging and testing your programs, ensure that your library list is changed to direct the programs to a test library containing test data so that any existing real data is not affected.

To prevent database files in production libraries from being modified unintentionally, do one of the following:

- Specify Start Debug (STRDBG) with the default **NO* for the *UPDPROD* parameter
- Use Change Debug (CHGDBG) with the default **NO* for the *UPDPROD* parameter
- Specify SET UPPROD NO in the Display Module Source display.

Debug Limits of the ILE Source Debugger

The ILE source debugger has the following limits:

- Function calls cannot be used in debug expressions. This is a limitation of the debug expression grammar.
- Precedence of operators and type conversion of mixed types conform to the C and C++ language standards.
- The maximum size of variables that can be displayed is 65535 characters:
 - With the *:c* and *:x* formatting overrides there are no exceptions.
 - With the *:s* formatting override, if no count is entered, the command stops after 30 bytes or a NULL, whichever is encountered first.

- With the `:f` formatting override, if no count is entered, the command stops after 1024 bytes or a NULL, whichever is encountered first.
- The maximum number of classes that can be inherited as virtual base classes in a single compilation unit is 512.

Preparing a Program for Debugging

A program or module must have debug data available if you are to debug it.

The type of debug data that can be associated with a module is referred to as a debug view.

The storage requirements for a module or program vary somewhat, depending on the type of debug data included. The debug options are listed below. Secondary storage requirements increase as you work down the list:

1. `DBGVIEW(*NONE)` (No debug data)
2. `DBGVIEW(*SOURCE)` (Source view)
3. `DBGVIEW(*LIST)` (Listing view)
4. `DBGVIEW(*STMT)` (Statement view)
5. `DBGVIEW(*ALL)` (All views).

Once you have created a module with debug data and bound it into a program (*PGM), you can start debugging.

Creating a Listing View

A **listing view** contains text similar to the text in the compiler listing produced by the compiler. You create a listing view to debug a module by using the `DBGVIEW(*LIST)` option when you create a module.

The compiler creates the listing view while the module (*MODULE) is being generated. The listing view is created by copying the text of the appropriate source members into the module. There is no dependency on the source members upon which it is based, once the listing view is created.

For example, to create a listing view to debug a program created from the source file `myfile.cpp`, type:

```
CRTBNDCPP MYFILE SRCSTMF('/home/myfile.cpp') DBGVIEW(*LIST)
```

Note: The maximum line length for a listing view is 255 characters.

Debug Commands

Many debug commands are available for use with the ILE source debugger. For example, if you type `break 10` on the debug command line, and press Enter, the Integrated Language Environment source debugger adds an unconditional breakpoint to line 10 of your source.

You can use these commands in the root source view, include source view, and listing view. They can also be used when the source is compiled with the statement view debug option. These options are: `DBGVIEW(*NONE)`, `DBGVIEW(*ALL)`, `DBGVIEW(*STMT)`, `DBGVIEW(*SOURCE)`, and `DBGVIEW(*LIST)`. **Debug data** is created when you compile a module with one of the debug options.

The commands and their parameters are entered on the Debug command line shown at the bottom of the Display Module Source display and the Evaluate Expression display. They can be entered in uppercase, lowercase, or mixed case. The online information describes the debug commands, and shows their allowed abbreviations.

The debug commands are as follows:

Command	Description
ATTR	Display the attributes of variables. The attributes are the size and type of the variable as recorded in the debug symbol table.
BREAK	Permits you to enter either an unconditional or conditional job breakpoint at a position in the program being tested. Use BREAK <i>line-number</i> WHEN <i>expression</i> to enter a conditional job breakpoint.
CLEAR	Remove conditional and unconditional breakpoints, or to remove one or all active watch conditions.
DISPLAY	Display the names and definitions assigned by using the EQUATE command. It also allows you to display a different source module than the one that is currently shown on the Display Module Source display. The module object must exist in the current program object.
EQUATE	Assign an expression, variable, or debug command to a name for shorthand use.
EVAL	Display or change the value of a variable or to display the value of expressions, records, structures, or arrays.
QUAL	Define the scope of variables that appear in subsequent EVAL or WATCH commands.
SET	Change debug options, such as the ability to update production files, specify if find operations are to be case sensitive, or to enable OPM source debug support.
STEP	Run one or more statements of the procedure that is being debugged.
TBREAK	Permits you to enter either an unconditional or conditional breakpoint in the current thread in a position in the program being tested.
THREAD	Allows you to display the Work with Debugged Threads display or change the current thread.
WATCH	Request a breakpoint when the contents of a specified storage location is changed from its current value.
FIND	<p>The find command searches in the module that is currently displayed for a specified line number or string of text. The text search can be specified in a forward or backward direction from the position of the cursor on the displayed view text. If the cursor is not on the view text the search starts at the first position of the top line of text on the current screen. When the string is successfully found, the cursor will be positioned on the first character of the found string.</p> <p>The last FIND command that is entered can be repeated by using the F16 Repeat Find key.</p>

UP	Moves the displayed window of source towards the beginning of the view by the amount that is entered.
DOWN	Moves the displayed window of source towards the end of the view by the amount that is entered.
LEFT	Moves the displayed window of source to the left by the number of characters that are entered.
RIGHT	Moves the displayed window of source to the right by the number of characters that are entered.
TOP	Positions the view to show the first line.
BOTTOM	Positions the view to show the last line.
NEXT	Positions the view to the next breakpoint in the source that is currently displayed.
PREVIOUS	Positions the view to the previous breakpoint in the source that is currently displayed.
HELP	Shows the online help information for the available source debugger commands.

Starting a Source Debug Session

You use the Start Debug (STRDBG) command to start the Integrated Language Environment source debugger. Once the debugger is started, it remains active until you enter the End Debug (ENDDBG) command.

Note: You must have *USE object authority to use the STRDBG command and *CHANGE authority for the objects that are to be debugged.

Initially you can add as many as twenty programs to a debug session by using the Program (PGM) parameter on the STRDBG command. They can be any combination of OPM or ILE programs. (Depending on how the OPM programs were compiled and also on the debug environment settings, you may be able to debug them by using the ILE source debugger.)

You can also add as many as twenty service programs to a debug session by using the Service program (SRVPGM) parameter on the STRDBG command.

Before you can debug an ILE C/C++ module with the Integrated Language Environment source debugger, you must compile the module using the CRTCMOD/CRTCPMOD command or the CRTBNDC/CRTBNDCPP command with one of the debug options *SOURCE, *LIST, *STMT, or *ALL.

You can create up to three views for each module that you want to debug. They are:

- Root source view

A **root source view** contains text from the root source member. This view does not contain any macro or include file expansions.

You can create a root source view by using the *SOURCE or *ALL options on the DBGVIEW parameter for either the CRTCMOD/CRTCPMOD or CRTBNDC/CRTBNDCPP commands when you create the module or the program, respectively.

The ILE C/C++ compiler creates the root source view while the module object (*MODULE) is being compiled. The root source view is created using references

to locations of text in the root source member rather than copying the text of the member into the view. For this reason, you should not modify, rename, or move root source members between the module creation of these members and the debugging of the module created from these members.

- Include source view

An **include source view** contains text from the root source member, as well as the text of all included members that are expanded into the text of the source. This view does not contain any macro expansion. When you use this view, you can debug the root source member and all included members.

For C only, you can create an include source view to debug a module by using the *SOURCE or *ALL option on the DBGVIEW parameter, and one of *SHOWINC, *SHOWSYS or *SHOWUSR on the OPTION parameter, depending on which include members you wish to add to the include source view when you create a module. For C++, OPTION(*SHOWINC || *SHOWSYS || *SHOWUSR) has the same effect on creation of the include source view.

The ILE C/C++ compiler creates the include view while the module object (*MODULE) is being compiled. The include view is created using references to locations of text in the source members (both root source member and included members) rather than copying the text of the members into the view. For this reason, you should not modify, rename, or move source and include members between the time the module object is created and the debugging of the module created from these members.

- Listing view

A **listing view** contains text similar to the compiler listing produced by the ILE C/C++ compiler specified in the OUTPUT() command parameter.

You can create a listing view to debug a module by using the *LIST or *ALL options when you compile the module. You can also specify at least one of *EXPMAC, *SHOWINC, *SHOWUSR, *SHOWSYS, and *SHOWSKP on the OPTION parameter, depending on the listing view that you want to see.

The first program that is specified on the STRDBG command is shown, if it has debug data and, if OPM, the OPMSRC parameter is *YES. If ILE and it has debug data, the entry module is shown; otherwise, the first module bound to the ILE program with debug data is shown.

1. The OPM program was compiled with OPTION(*LSTDBG) or OPTION(*SRCDBG). (Three OPM languages are supported: RPG, COBOL, and CL. RPG and COBOL programs must be compiled with *LSTDBG, or *SRCDBG, but CL programs must be compiled with *SRCDBG.)
2. The ILE debug environment is set to accept OPM programs. You can do this by specifying OPMSRC(*YES) on the STRDBG command. (The system default is OPMSRC(*NO).)

If these two conditions are not met, then debug the OPM program with the OPM system debugger.

To start a debug session for the sample debug program DEBUGEX which calls the OPM program RPGPGM, type:

```
STRDBG PGM(MYLIB/DEBUGEX MYLIB/RPGPGM) OPMSRC(*YES)
```

DBGVIEW(*NONE) is the default DBGVIEW option. No debug data is created when the module is created.

Once you have created a module with debug data or debug views, and bound it into a program object (*PGM), you can start to debug your program.

Example

This example shows you how to create three modules with debug views and start a debug session.

1. To create module T1520IC1 with a root source view, type:

```
CRTCMOD MODULE(MYLIB/T1520IC1) SRCFILE(QCPPL/QACSRC) DBGVIEW(*SOURCE)
```

A root source view and debug data is created to debug module T1520IC1.

2. To create module T1520IC2 with all three views, type:

```
CRTCMOD MODULE(MYLIB/T1520IC2) SRCFILE(QCPPL/QACSRC) DBGVIEW(*ALL)
OPTION(*SHOWINC)
```

All views and debug data are created to debug module T1520IC2.

3. To create module T1520IC3 with both root source and include view, type:

```
CRTCMOD MODULE(MYLIB/T1520IC3) SRCFILE(QCPPL/QACSRC) DBGVIEW(*SOURCE)
OPTION(*SHOWUSR)
```

An include view containing the root source member, user include files, and debug data is created to debug module T1520IC3.

4. To create program T1520PG1, type:

```
CRTPGM PGM(MYLIB/T1520PG1) MODULE(MYLIB/T1520IC1 MYLIB/T1520IC2) ENTMOD(*ONLY)
BNDDIR(MYLIB/T1520BD1) DETAIL(*FULL)
```

Note: The creation of this program requires modules, service programs, and a binding directory. See “Creating a Program in Two Steps” on page 24.

5. To start a debug session for program T1520PG1, type:

```
STRDBG PGM(MYLIB/T1520PG1)
```

The Display Module Source display appears as shown:

```

                                Display Module Source
Program:  T1520PG1      Library:  MYLIB      Module:  T1520IC1
1  /* This program demonstrates how to use multiple modules, service  */
2  /* programs and a binding directory. This program accepts user ID, */
3  /* item name, quantity and price, calculates the total cost and    */
4  /* writes an audit trail of the transaction.                        */
5
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <string.h>
9  #include <decimal.h>
10     11 int  calc_and_format (decimal(10,2),
11                               short int,
12                               char[];
13
14
15 void write_audit_trail (char[],

Debug . . . _____

F3=End program  F6=Add/Clear breakpoint  F10=Step  F11=Display variable
F12=Resume      F17=Watch variable      F18=Work with watch  F24=More keys
```

The module T1520IC1 is shown. It is the module with the main() function.

You can start a debug session for OPM programs or a combination of Integrated Language Environment and OPM programs by typing:

```
STRDBG PGM(MYLIB/T1520RP1 MYLIB/T1520CB1 MYLIB/T1520IC5) DSPMODSRC(*YES)
```


The parameter DSPMODSRC(*YES) specifies that you want the Integrated Language Environment source debug program display panel to be shown at start debug time. The DSPMODSRC parameter accepts the *PGMDEP value as a default. This value indicates that if any program in the STRDBG PGM list is an ILE program the source display panel is shown.

Adding and Removing Programs from a Debug Session

Programs and service programs can be added or removed from a debug session, after starting a debug session. You must have *CHANGE authority to a program to add it to or remove it from a debug session.

For ILE programs, use option 1 (Add program) on the Work with Module List display of the DSPMODSRC command. To remove an ILE program or service program, use option 4 (Remove program) on the same display. When an ILE program or service program is removed, all breakpoints for that program are removed. There is no limit to the number of ILE programs or service programs that can be included in a debug session at one time.

For OPM programs, you have two choices depending on the value specified for OPMSRC. If you specified OPMSRC(*YES), by using either STRDBG, the SET debug command, or Change Debug (CHGDBG) options, then you add or remove an OPM program using the Work With Module Display. (Note that there will not be a module name listed for an OPM program.) There is no limit to the number of OPM programs that can be included in a debug session when OPMSRC(*YES) is specified.

If you specified OPMSRC(*NO), then you must use the Add Program (ADDPGM) command or the Remove Program (RMVPGM) command. Only twenty OPM programs can be in a debug session at one time when OPMSRC(*NO) is specified.

Note: You cannot debug an OPM program with debug data from both an ILE and an OPM debug session. If OPM program is already in an OPM debug session, you must first remove it from that session before adding it to the ILE debug session or stepping into it from a call statement. Similarly, if you want to debug it from an OPM debug session, you must first remove it from an ILE debug session.

Example

This example shows you how to add an ILE C service program to, and remove an ILE C program from a debug session.

Note: Assume the ILE C program T1520ALP is part of this debug session, and the program has been debugged. It can be removed from this debug session.

1. To add programs to or remove programs from a debug session type:

DSPMODSRC

and press Enter. The Display Module Source display appears.

2. Press F14 (Work with module list) to show the Work with Module List display.
3. On this display type 1 (Add program) on the first line of the list to add programs and service programs to a debug session. To add service program T1520SP1, type: T1520SP1 for the *Program/module* field, MYLIB for the *Library* field, change the default program type from *PGM to *SRVPGM and press Enter.

4. On this display type 4 (Remove program) on the line next to each program or service program that you want to remove from the debug session.
5. To remove program T1520ALP, type: 4 next to T1520ALP, and press Enter.
6. Press F12 (Cancel) to return to the Display Module Source display.

If an ILE C/C++ program with debug data is in a debug session, the module with the `main()` function is shown (if it has a debug view). Otherwise, the first module bound to the ILE C/C++ program with debug data is shown.

Setting Debug Options

After you start a debug session, you can set or change the following debug options:

- Whether database files can be updated while debugging your program. (This option corresponds to the UPDPDPROD parameter of the STRDBG command.)
- Whether text searches using FIND are case sensitive.
- Whether OPM programs are to be debugged using the ILE source debugger. (This corresponds to the OPMSRC parameter.)

Changing the debug options using the SET debug command affects the value for the corresponding parameter, if any, specified on the STRDBG command. You can also use the Change Debug (CHGDBG) command to set debug options.

Example

This example shows you how to allow the ILE source debugger to add an OPM program to an ILE debug session.

Suppose you are in a debug session working with an ILE program and you decide you should also debug an OPM program that has debug data available. To enable the ILE source debugger to accept OPM programs, follow these steps:

1. After entering STRDBG, if the current display is not the Display Module Source display, type:

DSPMODSRC

The Display Module Source display appears

2. Type

SET

3. Press Enter. The Set Debug Options display appears. On this display type

Y

(Yes) for the OPM source debug support field, and press Enter to return to the Display Module Source display.

You can now add the OPM program, either by using the Work with Module display, or by processing a call statement to that program.

Example

This example shows how to set the Update production files debug option from a debug session.

1. To set debug options from a debug session type:

DSPMODSRC

and press Enter. The Display Module Source display appears.

2. Type
SET

and press Enter to show the Set Debug Options display.

3. On this display type Y (Yes) for the *Update production files* field, and press Enter to return to the Display Module Source display. The database files in production libraries are updated while the job is in debug mode.

Viewing the Program Source

The Display Module Source display shows the source of an ILE program object one module at a time. The source of an ILE module object can be shown if the module object was compiled using one of the following debug view options:

- DBGVIEW(*SOURCE)
- DBGVIEW(*COPY) - ILE RPG only
- DBGVIEW(*LIST)
- DBGVIEW(*ALL)

The source of an OPM program can be shown if the following conditions are met:

1. The OPM program was compiled with OPTION(*LSTDBG).
2. The ILE debug environment is set to accept OPM programs; that is the value of OPMSRC is *YES. (The system default is OPMSRC(*NO).)

Once you have displayed a view of a module, you may want to display a different module or see a different view of the same module (if you created the module with several different views). The ILE source debugger remembers that the last position in which the module is displayed, and displays it in the same position when a module is redisplayed. Lines that have breakpoints set are highlighted. When a breakpoint, step, or message causes the program to stop and the display to be shown, the statement where the breakpoint occurred is highlighted.

Displaying Other Modules in Your Program

You may want to set some debug options in other modules of your program. You can do this by changing the module that is shown on the Display Module Source display to specify the preferred module.

You can change the module that is shown on the Display Module Source display by using:

- The Work with Module list display
- The Display Module debug command

If you use this option with an ILE program object, the entry module with a root source, COPY, or listing view is shown (if it exists). Otherwise the first module object bound to the program object with debug data is shown. If you use this option with an OPM program object, then the source or listing view is shown (if available).

Example

This example shows you how to change from the module shown on the Display Module Source display to another module in the same program using Display Module debug command.

There are two types of breakpoints: job and thread. Each **thread** in a threaded application may have its own thread breakpoint at the same position at the same time. Both job and thread breakpoints can be unconditional or conditional. In general, there is one set of debug commands and Function keys for job breakpoints and another for thread breakpoints. For the rest of this section on breakpoints, the word breakpoint refers to both job and thread, unless specifically mentioned otherwise.

When the program stops, the Display Module Source display appears. Use this display to evaluate variables, set more breakpoints, and run any of the source debugger commands. The appropriate module is shown with the source positioned to the line where the condition occurred. The cursor will be positioned on the line where the breakpoint occurred if the cursor was in the text area of the display the last time the source was displayed. Otherwise, it is positioned on the debug command line.

If you change the view of the module after setting breakpoints, then the line numbers of the breakpoints are mapped to the new view by the source debugger.

You can set and remove unconditional and conditional breakpoints by using:

- F13 (Work with module breakpoints)
- F6 (Add/Clear breakpoint)

You can also add breakpoints with the BREAK or TBREAK debug commands. You can remove one, or all breakpoints from a program by using the Clear Program debug command.

To set a breakpoint on the first statement of a multi-statement macro, the cursor should be on the line containing the macro invocation, not the macro expansion.

Example

This example shows you how to set and remove an unconditional breakpoint using F6 (Add/Clear breakpoint), and add a conditional breakpoint using F13 (Work with module breakpoints).

1. To work with a module, type DSPMODSRC and press Enter. The Display Module Source display is shown.
If you want to set the breakpoint in the module shown, continue with step 3.
If you want to set a breakpoint in a different module, type display module name on the debug command line where name is the name of the module that you want to display.
2. Type display module T1520IC2, and press Enter.
3. To set an unconditional breakpoint, place the cursor on line 50.
4. Press F6 (Add/Clear breakpoint). A breakpoint is added to line 50 as shown:

```

Display Module Source
Program:  T1520PG1      Library:  MYLIB      Module:  T1520IC2
46      {
47      if (j<0) return(0);
48      if (hold_formatted_cost[i] == '$')
49      {
50          formatted_cost[j] = hold_formatted_cost[i];
51          break;
52      }
53      if (i<16 && !((i-2)%3))
54      {
55          formatted_cost[j] = ',';
56          --j;
57      }
58      formatted_cost[j] = hold_formatted_cost[i];
59      --j;
60      }
Debug . . . _____

F3=End program  F6=Add/Clear breakpoint  F10=Step  F11=Display variable
F12=Resume      F17=Watch variable  F18=Work with watch  F24=More keys
Breakpoint added to line 50

```

If there is no breakpoint on the line you specify, an unconditional breakpoint is set on that line.

Note: If there is a breakpoint on the line you specify, it is removed (even if it is a conditional breakpoint). This function key acts as a toggle key.

The clear command can be used to remove a breakpoint. For example, clear 50 removes the breakpoint at line 50.

5. To set a conditional breakpoint press F13 (Work with module breakpoints). The Work with Module Breakpoints display is shown.
6. On this display type 1 (Add) on the first line of the list to add a conditional breakpoint.
7. To set a conditional breakpoint at line 35 when i is equal to 21, enter 35 for the *Line* field, i==21 for the *Condition* field, and press Enter as shown:

```

Work with Module Breakpoints
Program . . . : T1520PG1      Library . . . : MYLIB      System:  TORASD80
Module . . . : T1520IC2      Type . . . . : *PGM
Type options, press Enter.
1=Add  4=Clear
Opt   Line   Condition
1     35     i==21
_     50

```

A conditional breakpoint is set on line 35. The expression is evaluated before the statement is run. If the result is true (in the example, if i is equal to 21), the program stops, and the Display Module Source display is shown. If the result is false, the program continues to run. If you do not want to switch panels, you can set the same breakpoint from the Display Module Source command line by typing:

```
break 35 when i==21
```

You can set a conditional breakpoint to a statement. For example, if you have a compiler listing that contains line numbers and statement numbers, you can use the statement syntax to set a breakpoint on a specific statement when there are several statements on a single line.

Line	Stmt	Source
33	24	i=j; j=0;
34	26	array[i] = cost;

Break myfunction/25 sets a breakpoint on the statement j=0 assuming this is in myfunction. If you then enter Break 33, a breakpoint is set at statement 24, i=j.

An existing breakpoint is always replaced by a new breakpoint entered at the same location.

8. Repeat steps 4 or step 5 for each breakpoint that you want to add.
9. After the breakpoints are set, press F12 (Cancel) to leave the Work with Module Breakpoints display. Press F3 (End Program) to leave the ILE source debugger. Your breakpoints are not removed.
10. Call the program. When a breakpoint is reached, the program stops, and the Display Module Source display is shown again.

Setting and Removing Unconditional Breakpoints

You can set or remove an unconditional breakpoint by using:

- F6 (Add/Clear breakpoint) from the Display Module Source display
- F13 (Work with module breakpoints) from the Display Module Source display
- The BREAK debug command to set a breakpoint
- The CLEAR debug command to remove a breakpoint

The simplest way to set and remove an unconditional breakpoint is to use F6 (Add/Clear breakpoint). The function key acts as a toggle. If a breakpoint is already set on the current line, pressing F6 removes it while the cursor is positioned on the current line.

To set or remove an unconditional breakpoint from the Display Module Source display press F13 (Work with module breakpoints). A list of options appear that allow you to set or remove breakpoints. If you select 4 (Clear), a breakpoint is removed from the line.

An alternate method of setting and removing unconditional breakpoints is to use the BREAK and CLEAR debug commands. To set an unconditional breakpoint using the BREAK debug command, type `BREAK line-number` on the debug command line. The variable *line-number* is the line number in the currently displayed view of the module on which you want to set a breakpoint.

To remove an unconditional breakpoint using the CLEAR debug command, type:
`CLEAR line-number`

on the debug command line.

Setting and Removing Conditional Thread Breakpoints

You can set or remove a conditional thread breakpoint by using:

- The Work with Module Breakpoints display
- The TBREAK debug command to set a conditional thread breakpoint in the current thread
- The CLEAR debug command to remove a conditional thread breakpoint.

Using the Work with Module Breakpoints Display

To set a conditional thread breakpoint using the Work with Module Breakpoints display:

1. Press F13 to display Work with Module Breakpoints.
2. Type 1 (Add) in the *Opt* field.
3. Fill in the remaining fields as if it were a conditional job breakpoint.
4. Press Enter.

To remove a conditional thread breakpoint using the Work with Module Breakpoints display:

1. Type 4 (Clear) in the *Opt* field next to the breakpoint you want to remove.
2. Press Enter.

Using the TBREAK or CLEAR Debug Commands

You use the same syntax for the TBREAK debug command as you would for the BREAK debug command. The difference between these commands is that the BREAK debug command sets a conditional job breakpoint at the same position in all threads, while the TBREAK debug command sets a conditional thread breakpoint in the current thread.

To remove a conditional thread breakpoint, use the CLEAR debug command. When a conditional thread breakpoint is removed, it is removed for the current thread only.

Setting a Conditional Breakpoint Using F13

To set a conditional breakpoint using F13 (Work with module breakpoints) do the following:

1. Press F13 (Work with module breakpoints) from the Display Module Source window. The Work with Module Breakpoints display is shown.
2. On this display, type 1 (Add) on the first line of the list to add a conditional breakpoint.
3. To set a conditional breakpoint at line 35 when the variable *i* is equal to 21, type 35 for the **Line** field, *i*==21 for the **Condition** field, and press Enter.

A conditional breakpoint is set on line 35. The expression is evaluated before the statement is run. If the result is true (in the example, if *i*==21), the program stops, and the Display Module Source display is shown. If the result is false, the program continues to run.

An existing breakpoint is always replaced by a new breakpoint entered at the same location.

4. After the breakpoint is set, press F12 (Cancel) to leave the Work with Module Breakpoints display. Press F3 (End Program) to leave the ILE source debugger. Your breakpoint is not removed.
5. Call the program. When the conditional breakpoint is reached and the condition is true, the program stops, and the Display Module Source display is shown again. At this point, you can step through the program or resume processing.

Setting a Conditional Breakpoint Using the BREAK Command

Assume you want to stop the program when the variable *j* has a certain value. To specify the conditional breakpoint using the BREAK command do the following:

1. From the Display Module Source display, enter break 56 when j==5 to set a conditional breakpoint on line 56.
2. After the breakpoint is set, press F3 (End Program) to leave the ILE source debugger. Your breakpoint is not removed.
3. Call the program. When a breakpoint is reached, the program stops, and the Display Module Source display is shown again.

Removing All Breakpoints

You can remove all breakpoints, conditional and unconditional, from a program that has a module shown on the Display Module Source display by using the CLEAR PGM debug command. To use the debug command, type CLEAR PGM on the debug command line. The breakpoints are removed from all of the modules bound to the program.

Setting and Removing Watch Conditions

Use a **watch condition** to monitor changes in the current value of a variable or an expression which determines the address of a storage location. Setting watch conditions is similar to setting conditional breakpoints, with one important difference:

- Watch conditions stop the program as soon as the value of a variable changes from its current value.
- Conditional breakpoints stop the program only if a variable changes to the value specified in the condition.

The debugger watches a variable through the content of a **storage address**, computed at the time the watch condition is set. When the content at the storage address is changed from the value it had when the watch condition was set or when the last watch condition occurred, a breakpoint is set, and the program stops.

Note: After a watch condition has been registered, the new content at the watched storage location is saved as the new current value of the corresponding variable. The next watch condition will be registered if the new content at the watched storage location changes subsequently.

Characteristics of Watches

When using watches, keep the following watch characteristics in mind:

- Watches are monitored on a system-wide basis, with a maximum number of 256 watches that can be active simultaneously. This number includes watches set by the system.

Depending on overall system use, you may be limited in the number of watch conditions you can set at a given time. If you try to set a watch condition while the maximum number of active watches across the system is exceeded, you will receive an error message and the watch condition is not set.

Note: If a variable crosses a page boundary, two watches are used internally to monitor the storage locations. Therefore, the maximum number of variables that can be watched simultaneously on a system-wide basis ranges from 128 to 256.

- Watch conditions can only be set when a program is stopped under debug, and the variable to be watched is in scope. If this is not the case, an error message is issued when a watch is requested, indicating that the corresponding call stack entry does not exist.

- Once the watch condition is set, the address of a storage location that is watched does not change. Therefore, if a watch is set on a temporary location, it could result in spurious watch-condition notifications.

An example of this is the automatic storage of an ILE C or C++ procedure, which can be re-used after the procedure ends.

A watch condition may be triggered even though the watched variable is no longer in scope. You must not assume that a variable is in scope just because a watch condition has been reported.

- Two watch locations in the same job must not overlay in any way. Two watch locations in different jobs must not start at the same storage address; otherwise, overlap is allowed. If these restrictions are violated, an error message is issued.

Note: Changes that are made to a watched storage location are ignored if they are made by a job other than the one that set the watch condition.

- After the command is successfully run, your application is stopped if a program in your session changes the content of the watched storage location, and the Display Module Source display is shown.

If the program has debug data, it will be shown if a source view is available. The source line of the statement that was about to be run when the content change at the storage-location was detected is highlighted. A message indicates which watch condition was satisfied. If the program cannot be debugged, the text area of the display will be blank.

- Eligible programs are automatically added to the debug session if they cause the watch-stop condition.
- When multiple watch conditions are hit on the same program statement, only the first one will be reported.
- You can set watch conditions when you are using service jobs for debugging, that is when you debug one job from another job.

Setting Watch Conditions

Your program must be stopped under debug, and the variable you want to watch must be in scope before you can set a watch condition:

- To watch a global variable, you must ensure that the program in which the variable is defined is active before setting the watch condition.
- To watch a local variable, you must step into the function in which the variable is defined before setting the watch condition.

You can set a watch condition by using:

- F17 (watch variable) to set a watch condition for the variable under the cursor.
- The WATCH debug command with or without its parameters.

Using the WATCH Command

If you use the WATCH command, it must be entered as a single command; no other debug commands are allowed on the same command line.

- To access the Work with Watch display shown below, type WATCH on the debug command line, without any parameters.

The scope of the expression variables in a watch is defined by the most recently issued QUAL command.

- To set a watch condition and specify a watch length, type: WATCH expression : watch-length on a debug command line.

Each watch allows you to monitor and compare a maximum of 128 bytes of contiguous storage. If the maximum length of 128 bytes is exceeded, the watch condition will not be set, and the debugger issues an error message.

By default, the length of the expression type is also the length of the watch-comparison operation. The watch-length parameter overrides this default. It determines the number of bytes of an expression that should be compared to determine if a change in value has occurred.

For example, if a 4-byte binary integer is specified as the variable, without the watch-length parameter, the comparison length is four bytes. However, if the watch-length parameter is specified, it overrides the length of the expression in determining the watch length.

Displaying Active Watches

To display a system-wide list of active watches and show which job set them, type DSPDBGWCH on a CL command line. This command brings up the Display Debug Watches display that is shown below.

Display Debug Watches					
-----Job-----			NUM	LENGTH	System: DEBUGGER ADDRESS
MYJOBNAME1	MYUSERPRF1	123456	1	4	080090506F027004
JOB4567890	PRF4567890	222222	1	4	09849403845A2C32
JOB4567890	PRF4567890	222222	2	4	098494038456AA00
JOB	PROFILE	333333	14	4	040689578309AF09
SOMEJOB	SOMEPROFIL	444444	3	4	005498348048242A
Bottom					
Press Enter to continue					
F3=Exit F5=Refresh F12=Cancel					

Note: This display does not show watch conditions that are set by the system.

Removing Watch Conditions

Watches can be removed in the following ways:

- The CLEAR command that is used with the WATCH keyword selectively ends one or all watches. For example, to clear the watch that is identified by watch-number, type:

```
CLEAR WATCH watch-number
```

The watch number can be obtained from the Work with Watches display.

To clear all watches for your session, type:

```
CLEAR WATCH ALL
```

Note: While the CLEAR PGM command removes all breakpoints in the program that contains the module being displayed, it has no effect on watches. You must explicitly use the WATCH keyword with the CLEAR command to remove watch conditions.

- The CL End Debug (ENDDBG) command removes watches that are set in the local job or in a service job.

Note: ENDDDBG will be called automatically in abnormal situations to ensure that all affected watches are removed.

- The initial program load (IPL) of your iSeries system removes all watch conditions system-wide.

Example of Setting a Watch Condition

In this example, you watch a variable salary in program MYLIB/PAYROLL. To set the watch condition, type WATCH salary on a debug line, accepting the default value for the watch-length.

If the value of the variable salary changes subsequently, the application stops, and the Display Module Source display is as shown:

```
Display Module Source
Program:  PAYROL      Library:  MYLIB      Module:  PAYROLL
52  for  (cnt=0;
53      cnt<EMPMAX &&;
54      scanf("%s%s%f%d%d", payptr->first, payptr->last,
55          &(payptr->wage), &eflag, &(payptr->hrs))!=EOF;
56      cnt++, payptr++)
57  {
58      payptr->exempt=eflag;
59  }
60  empsort(payfile, cnt);
61  for  (index=1, payptr=payfile; index<=cnt; index++,payptr++) {
62      if  (payptr->exempt==1) {
63          salary = 40*(payptr->wage);
64          numexempt++; }
65      else
66          salary = (payptr->hours)*(payptr->wage);
More...
```

Debug . . . _____

F3=End program F6=Add/Clear breakpoint F10=Step F11=Display variable
F12=Resume F17=Watch variable F18=Work with watch F24=More keys
Watch number 1 at line 64, variable: salary

- The line number of the statement where the change to the watch variable was detected is highlighted. This is typically the first executable line *following* the statement that changed the variable.
- A message indicates that the watch condition was satisfied.

If a text view is not available, a blank Display Module Source display is shown, with the same message as above in the message area.

```
Display Module Source
(Source not available)

F3=End program F12=Resume F14=Work with module list F18 Work with watch
F21=Command entry F22=Step into F23=Display output
Watch number 1 at instruction 18, variable: salary
```

The following programs cannot be added to the ILE debug environment:

1. ILE programs without debug data
2. OPM programs with non-source debug data only
3. OPM programs without debug data

In the first two cases, the stopped statement number is passed. In the third case, the stopped MI instruction is passed. The information is displayed at the bottom of a blank Display Module Source display as shown above. Instead of the line number, the statement or the instruction number is given.

Stepping through the Program

The step function of the ILE source debugger allows you to run a specified number of statements of a program, and then return to the Display Module Source display at the position of the next statement to be run. The cursor is positioned on this statement if the cursor was in the text area of the display the last time the source was displayed. Otherwise, it is positioned on the debug command line. The program begins at the statement where the program stopped. Setting a breakpoint causes the program to stop before the statement is run. The default number of statements to run is one.

When calls to other functions are encountered, you can step into an OPM program if it has debug data available and if the debug session accepts OPM programs for debugging.

If the ILE source debugger is not set to accept OPM programs, or if there is no debug data available, then you see a blank Display Module Source display with a message indicating that the source is not available. (An OPM program has debug data if it was compiled with `OPTION(*LSTDBG)`.)

The default step mode is step over.

Stepping over Programs

You can step over programs by using:

- F10 (Step) on the Display Module Source display
- The STEP OVER debug command.

Using F10 to Step over Programs

Use F10 (Step) on the Display Module Source display to step over a called program in a debug session. If the next statement to be run is a CALL statement to another program, pressing F10 (Step) causes the called program to run to completion before the calling program is stopped again.

Using the STEP OVER Debug Command

Use the STEP OVER debug command to step over a called program in a debug session. To use the STEP OVER debug command, type `STEP number-of-statements OVER` on the debug command line. The variable *number-of-statements* is the number of statements of the program that you want to run in the next step before the program is halted again.

If this variable is omitted, the default is 1. If one of the statements that are run contains a call to another program, the ILE source debugger steps over the called program.

Stepping into Programs

Step into programs by using:

- F22 (Step into) on the Display Module Source display
- The STEP INTO debug command.

Using F22 to Step into Programs

Use F22 (Step into) on the Display Module Source display to step into a called program in a debug session. If the next statement to be run is a CALL statement to another program, pressing F22 causes the first executable statement in the called program to be run. The called program is then shown in the Display Module Source display.

Note: The called program must have debug data associated with it in order for it to be shown in the Display Module Source display.

Using the STEP INTO Debug Command

Use the STEP INTO debug command to step into a called program in a debug session. To use the STEP INTO debug command, type:

```
STEP number-of-statements INTO
```

on the debug command line. The variable *number-of-statements* is the number of statements of the program that you want to run in the next step before the program is halted again. If this variable is omitted, the default is 1.

Stepping into Called Programs

If one of the statements that are run contains a CALL statement to another program, the source debugger steps into the called program. Each statement in the called program is counted in the step. If the step ends in the called program, the called program is shown in the Display Module Source display. For example, if you type STEP 5 INTO on the debug command line, the next five statements of the program are run. If the third statement is a CALL statement to another program, two statements of the calling program are run and the first three statements of the called program are run.

The STEP INTO command works with the CL CALL command as well. You can take advantage of this to step through your program after calling it. After starting the source debugger, from the initial Display Module Source display, enter STEP 1 INTO. This sets the step count to 1. Use the F12 key to return to the command line and then call the program. The program stops at the first statement with debug data.

Stepping Into a Program Using F22

Use F22 (Step Into) to step into program CPGM from the program DEBUGEX.

1. Assume that the Display Module Source Display shows the source for DEBUGEX
2. To set an unconditional breakpoint at line 92, which is the last executable statement before the call to function CalcTax() in program CPPPGM, type Break and press Enter.
3. Press F3 (End Program) to leave the Display Module Source display.
4. Call the program. The program stops at breakpoint 92, as shown in Figure 36 on page 108.

DEBUGEX Before Stepping Into CPGM

```

Display Module Source
Program:  DEBUGEX      Library:  MYLIB      Module:  DEBUGEX
88      cout << "Please enter amount" << endl;
89      cin >> input;
90      if (input > MINIMUM) {
91          // call function CalcTax in separate program CPPPGM
92          retval1 = CalcTax(input);
93          if (retval1 > LIMIT)
94              retval2 = CalcSurtax(input)
95      }
96      cout << "Total tax is " << retval1 = retval2 << endl;
97  }
98
99
100
101
102
More...

Debug . . .

```

```

F3=End program  F6=Add/Clear breakpoint  F10=Step  F11=Display variable
F12=Resume      F17=Watch variable      F18=Work with watch  F24=More keys
Breakpoint at line 90

```

Figure 36. Module Source Display for DEBUGEX

- Press F22 (Step into). One statement of the program runs, and then the Display Module Source display of CPGM is shown.

The first executable statement of CPGM is processed (line 13) and then the program stops.

Note: You cannot specify the number of statements to step through when you use F22. Pressing F22 performs a single step.

```

Display Module Source
Program:  CPGM      Library:  MYLIB
1  =====
2  * CPGM - Program called by DEBUGEX to illustrate the
3  * STEP functions of the ILE source
4  *debugger
5  * This program receives a parameter input from DEBUGEX,
6  * calculates a tax amount, and then returns
7  =====
8
9  double CalcTax(double input)
10 {
11     double tax;
12
13     tax= input * TAXRATE
14
15     return taxrate;
Bottom

Debug . . .

```

```

F3=End program  F6=Add/Clear breakpoint  F10=Step  F11=Display variable
F12=Resume      F17=Watch variable      F18=Work with watch  F24=More keys
Step completed at line 13.

```

Figure 37. Module Source Display After Stepping Into CPGM

If there is no debug data available, you see a blank Display Module Source display with a message indicating that the source is not available.

Stepping over Procedures

If you specify over on the step debug command, calls to procedures and functions count as single statements. This is the default step mode. Stepping through four statements of a program could result in running 20 statements if one of the four is a call to a procedure with 16 statements. You can start the step-over function by using:

- The Step Over debug command
- F10 (Step)

Example

This example shows you how to use F10 (Step) to step over one statement at a time in your program.

1. To work with a module type DSPMODSRC and press Enter. The Display Module Source display is shown.
2. Type display module T1520IC2, and press Enter.
3. To set an unconditional breakpoint at line 50, type Break 50 on the debug command line, and press Enter.
4. To set a conditional breakpoint at line 35, type Break 35 when i==21 on the debug command line, and press Enter.
5. Press F12 (Resume) to leave the Display Module Source display.
6. Call the program. The program stops at breakpoint 35 if i is equal to 21, or at line 50 whichever comes first.
7. To step over a statement, press F10 (Step). One statement of the program runs, and then the Display Module Source display is shown. If the statement is a function call, the function runs to completion. If the called function has a breakpoint set, however, the breakpoint will be hit. At this point you are in the function and the next step will take you to the next statement inside the function.

Note: You cannot specify the number of statements to step through when you use F10. Pressing F10 performs a single step.

```

                                Display Module Source
Program:  T1520PG1      Library:  MYLIB      Module:  T1520IC2
 47      if (j<0) return(0);
 48      if (hold_formatted_cost[i] == '$')
 49      {
 50          formatted_cost[j] = hold_formatted_cost[i];
 51          break;
 52      }
 53      if (i<16 && !((i-2)%3))
 54      {
 55          formatted_cost[j] = ',';
 56          --j;
 57      }
 58      formatted_cost[j] = hold_formatted_cost[i];
 59      --j;
 60      }
 61
Debug . . . _____

F3=End program  F6=Add/Clear breakpoint  F10=Step  F11=Display variable
F12=Resume      F17=Watch variable      F18=Work with watch  F24=More keys
Breakpoint at line 50.
```

8. To step over 5 statements, type step 5 over on the debug command line, and press Enter. The next five statements of your program run, and then the Display Module Source display is shown.
If the third statement is a call to a function, the first two statements run, the function is called and returns, and the last two statements run.
9. To step over 11 statements, type step 11 over on the debug command line, and press Enter. The next 11 statements of your program run. The Display Module Source display is shown.

Stepping into Procedures

There is an automatic feature for stepping. This feature automatically puts a service program into debug. This happens if the service program that is stepped into from another program in debug:

- Has debug data
- Is not in debug
- Contains a procedure

The service program is added to debug for the user, and the DSPMODSRC panel shows the procedure in the service program. From this point, modules in the service program can be accessed using the Work with Modules display just like modules in programs the user added to debug.

If you specify *into* on the step debug command, each statement in a procedure or function that is called counts as a single statement. You can start the step into function by using:

- The Step Into debug command
- F22 (Step into)

Example

This example shows you how to use F22 (Step into) to step into one procedure.

1. Type DSPMODSRC and press Enter. The Display Module Source display is shown.
2. To set an unconditional breakpoint at line 50, type Break 50 on the debug command line, and press Enter.
3. To set a conditional breakpoint at line 35, type Break 35 when i==21 on the debug command line, and press Enter.
4. Press F12 (Resume) to leave the Display Module Source display.
5. Call the program. The program stops at breakpoint 35 if i is equal to 21 or at line 50 whichever comes first.

```

                                Display Module Source
Program:  T1520PG1      Library:  MYLIB      Module:  T1520IC2
47      if (j<0) return(0);
48      if (hold_formatted_cost[i] == '$')
49      {
50          formatted_cost[j] = hold_formatted_cost[i];
51          break;
52      }
53      if (i<16 && !((i-2)%3))
54      {
55          formatted_cost[j] = ',';
56          --j;
57      }
58      formatted_cost[j] = hold_formatted_cost[i];
59      --j;
60      }
61
Debug . . . _____

F3=End program  F6=Add/Clear breakpoint  F10=Step  F11=Display variable
F12=Resume      F17=Watch variable      F18=Work with watch  F24=More keys
Breakpoint at line 50.

```

6. Press F22 (Step into). One statement of the program runs, and then the Display Module Source display is shown. If the statement is a procedure or function call, only the first statement of the procedure or function runs.

Note: You cannot specify the number of statements to step through when you use F22. Pressing F22 performs a single step.

7. To step into 5 statements, type step 5 into on the debug command line, and press Enter.
The next five statements of your program are run, and then the Display Module Source display is shown. If the third statement is a call to a function, the first two statements of the calling procedure run, and the first three statements of the function run.
8. To step into 11 statements, type step 11 into on the debug command line, and press Enter. The next 11 statements of your program runs. The Display Module Source display is shown.

Displaying or Changing the Value of Variables

You can display the value of scalar variables, expressions, structures, arrays, or errno and change the value of scalar variables and errno using the eval debug command. The module that is shown on the Display Module Source display must be bound to a program that is in a debug session. The scope of the variables used in the eval debug command is defined by using the qual debug command. The program must be called and stopped at a breakpoint or step location to display or change the value.

- The eval debug command
- F11 (Display variable)

You can use the Enter key as a toggle switch between displays.

You can change variables by using the eval debug command with assignment.

Example

This example shows you how to use the F11 (Display variable) to display a variable.

1. Type DSPMODSRC and press Enter. The Display Module Source display is shown.
2. Type display module T1520IC2, and press Enter.
3. Place the cursor on the variable hold_formatted_cost on line 50 and press F11 (Display variable). A pointer to the array is shown on the message line in the following.

```

                                Display Module Source
Program:  T1520PG1      Library:  MYLIB      Module:  T1520IC2
 47      if (j<0) return(0);
 48      if (hold_formatted_cost[i] == '$')
 49      {
 50          formatted_cost[j] = hold_formatted_cost[i];
 51          break;
 52      }
 53      if (i<16 && !((i-2)%3))
 54      {
 55          formatted_cost[j] = ',';
 56          --j;
 57      }
 58      formatted_cost[j] = hold_formatted_cost[i];
 59      --j;
 60      }
 61
More...

Debug . . . _____

F3=End program  F6=Add/Clear breakpoint  F10=Step  F11=Display variable
F12=Resume      F17=Watch variable      F18=Work with watch  F24=More keys
hold_formatted_cost = SPP:C048BD0003F0

```

Messages with multiple line responses will cause the Evaluate Expression display to be shown. This display will show all response lines. It also shows a history of the debug commands entered and the results from these commands. To return to the Display Module Source display, press the ENTER key. You can use the Enter key as a toggle switch between displays. Single-line responses will be shown on the Display Module Source message line.

You can also use the eval debug command to determine the value of an expression. For example, if j has a value of 1024, type eval (j * j)/512 on the debug command line. You use the qual debug command to determine the line or statement number within the function that you want the variables scoped to for the eval debug command. The Evaluate Expression display shows (j * j)/512 = 2048.

Example

This example shows you how to use the eval debug command to assign an expression to a variable.

1. Type DSPMODSRC and press Enter. The Display Module Source display is shown.
2. Type display module T1520IC2, and press Enter.
3. To specify the scope of the eval command you can use a qualify command. For example, qual 48. will qualify the eval command to the scope that line 48 is located at. Line 48 is the number within the function to which you want the variables scoped for the following eval debug command.

Note: You do not always have to use the qual debug command before the eval debug command. An automatic qual is done when a breakpoint is encountered or a step is done. This establishes the default for the scoping rules to be the current stop location.

4. To change an expression in the module shown type: eval x=<expr> where x is the variable name and <expr> is the expression you want to assign to variable x.

For example, "eval hold_formatted_cost [1] = '#'" changes the array element at 1 from \$ to # and shows "hold_formatted_cost[1]= '#' = '#':" on the Display Module Source display as shown:

Display Module Source

Program: T1520PG1 Library: MYLIB Module: T1520IC2

47 if (j<0) return(0);

48 if (hold_formatted_cost[i] == '\$')

49 {

50 formatted_cost[j] = hold_formatted_cost[i];

51 break;

52 }

53 if (i<16 && !((i-2)%3))

54 {

55 formatted_cost[j] = ',';

56 --j;

57 }

58 formatted_cost[j] = hold_formatted_cost[i];

59 --j;

60 }

61

Debug . . . _____

F3=End program F6=Add/Clear breakpoint F10=Step F11=Display variable

F12=Resume F17=Watch variable F18=Work with watch F24=More keys

hold_formatted_cost[1]= '#' = '#'

Changing the Value of Scalar Variables

Change the value of scalar variables using the EVAL debug command with an assignment operator (=). The program must be called and stopped at a breakpoint or step location to change the value. To change the value of a variable, type:

EVAL variable-name = value

on the debug command line. variable-name is the name of the variable that you want to change and value is an identifier, literal, or constant value that you want to assign to variable *variable-name*.

EVAL COUNTER=3

changes the value of COUNTER to 3 and shows

COUNTER=3 = 3

on the message line of the **Display Module Source** display.

Use the EVAL debug command to assign numeric, alphabetic, and alphanumeric data to variables. When you assign values to a character variable, the following rules apply:

- If the length of the source expression is less than the length of the target expression, the data is left justified in the target expression and the remaining positions are filled with blanks
- If the length of the source expression is greater than the length of the target expression, the data is left justified in the target expression and truncated to the length of the target expression.

The scope of the variables used in the EVAL debug command is defined by using the QUAL debug command. To change a variable at line 48, type QUAL 48. Line 48 is the number within a function to which you want the variables scoped for the EVAL debug command.

Note: You do not always have to use the QUAL debug command before the EVAL debug command. An automatic QUAL is done when a breakpoint is encountered or a step is done. This establishes the default for the scoping rules to be the current stop location.

The example below shows the results of changing the array element at 1 from \$ to #.

```
EVAL hold_formatted_cost [1] = '#'
      hold_formatted_cost[1]= '#' = '#':

//Code evaluated before statement 51 where a breakpoint is set
47     if (j<0) return(0);
48     if (hold_formatted_cost[i] == '$')
49     {
50         formatted_cost[j] = hold_formatted_cost[i];
51         break;
52     }
53     if (i<16 && !((i-2)%3))
54     {
55         formatted_cost[j] = ',';
56         --j;
57     }
58     formatted_cost[j] = hold_formatted_cost[i];
59     --j;
60 }
61
```

Figure 38. Using EVAL to Change a Variable

Equating a Name with a Variable, Expression, or Command

You can equate a name with a variable, expression, or debug command for shorthand use. You can then use that name alone or within another expression. If you use it within another expression, the value of the name is determined before the expression is evaluated. Equates stay active until a debug session ends or a name is removed.

Example

This example shows you how to use the equate debug command with a variable name.

1. Type DSPMODSRC and press Enter. The Display Module Source display is shown.
2. To equate an expression, type `equate <name> <definition>` where <name> is a character string that contains no blanks and <definition> is a character string separated from <name> by at least one blank. The character strings can be in uppercase, lowercase, or mixed case. The length of the character strings combined is limited to 144 characters, which is the length of the command line. After any equates have been expanded, the length is limited to 150 characters, which is the maximum command length. For example, type `equate dv display variable`.

If a definition is not supplied, and a previous equate debug command has defined the name, the previous definition is removed. If the name was not previously defined, an error message is shown.

To see the equates that are defined for this debug session, type: display equate. A list of the active equates is shown on the Evaluate Expression display.

Displaying a Structure

The following example shows a structure with two elements being displayed. Each element of the structure is formatted according to its type and displayed.

1. Type DSPMODSRC, and press Enter. The Display Module Source display is shown.
2. Set a breakpoint at line 9.
3. Press F12 (Resume) to leave the Display Module Source display.
4. Call the program. The program stops at the breakpoint at line 9.
5. Type eval test on the debug command line, and press Enter as shown:

Display Module Source

Program: TEST1 Library: DEBUG Module: MAIN
1 struct {
2 char charValue;
3 unsigned long intValue;
4 } test;
5
6 main(){
7 test.intValue = 10;
8 test.charValue = 'c';
9 test.charValue = 11;
10 }

Bottom

Debug . . . eval test

F3=Exit program F6=Add/Clear breakpoint F10=Step F11=Display variable
F12=Resume F17=Watch variable F18=Work with watch F24=More keys

6. Press Enter to go to the next display. The Evaluate Expression Display shows the entire structure as shown:

Evaluate Expression

Previous debug expressions
> BREAK 9
> EVAL test
test.charValue = 'c'
test.intValue = 10

7. Press Enter from the Evaluate Expression Display to return to the Display Module Source screen.

Displaying Variables as Hexadecimal Values

The following example shows the steps and syntax used to dump hexadecimal variables.

1. Type DSPMODSRC, and press Enter. The Display Module Source display appears, as shown below.
2. Set a breakpoint at line 9.
3. Press F12 (Resume) to leave the Display Module Source display.
4. Call the program. The program stops at the breakpoint at line 9.
5. Type eval test: x 32 on the debug command line, and press Enter as shown below.

```

Display Module Source
Program:  TEST1      Library:  DEBUG      Module:  MAIN
1  struct {
2  char charValue;
3  unsigned long intValue;
4  } test;
5
6  main(){
7  test.intValue = 10;
8  test.charValue = 'c';
9  test.charValue = 11;
10 }

Debug . . . eval test: x 32_____ Bottom

F3=Exit program  F6=Add/Clear breakpoint  F10=Step  F11=Display variable
F12=Resume       F17=Watch variable      F18=Work with watch  F24=More keys

```

- The Evaluate Expression display appears. As requested, 32 bytes are shown, but only the first 8 bytes are meaningful. The left column is an offset in hex from the start of the variable. The right column is an EBCDIC character representation of the data. If no length is specified after the 'x', the size of the variable is used as the length. A minimum of 16 bytes is displayed. Press the Enter key to return to the Display Module Source display.

```

Evaluate Expression

Previous debug expressions

> BREAK 9
> EVAL test: x 32
00000  83000000 0000000A 00000000 00000000 - C.....
00010  00000000 00000000 00000000 00000000 - .....

```

Displaying Null Ended Character Arrays

The following example shows the display of a character string. The array must be dereferenced by the '*' operator. If the * operator is not entered, the array is displayed as a space pointer. If the dereferencing operator is used, but the 's' is not appended to the expression, only the first array element is displayed.

- While in a debug session, type DSPMODSRC, and press Enter. The Display Module Source display is shown.
- Set a breakpoint at line 6.
- Press F12(Resume) to leave the Display Module Source Display.
- Call the program. The program stops at the breakpoint at line 6.
- Type eval *array1: s on the debug command line, and press Enter as shown:

```

Display Module Source
Program:  TEST3      Library:  DEBUG      Module:  MAIN
1  #include <string.h>
2  char array1 [11];
3  int i;
4  main(){
5  strcpy(array1,"0123456789");
6  i = 0;
7  }

Debug . . . eval *array1: s_____ Bottom

F3=Exit program  F6=Add/Clear breakpoint  F10=Step  F11=Display variable
F12=Resume       F17=Watch variable      F18=Work with watch  F24=More keys

```


The following shows the value of the array. A string length of up to 65535 can follow the s character. Formatting will stop at the first null character encountered. If no length is specified, formatting will stop after 30 characters or the first null, whichever is less.

```

                                Display Module Source
Program:  TEST3                Library:  DEBUG                Module:  MAIN
1  #include <string.h>
2  char array1 [11];
3  int i;
4  main(){
5      strcpy(array1,"0123456789");
6      i = 0;
7  }

                                Bottom
Debug . . .

F3=Exit program  F6=Add/Clear breakpoint  F10=Step  F11=Display variable
F12=Resume       F17=Watch variable       F18=Work with watch  F24=More keys
*array1: s = "0123456789"

```

The following example shows the usage of the 'f' syntax to specify that the newline character (x'15') should be scanned for while displaying string output. If the end of the display line occurs, the output is wrapped to the next display line.

When the :f formatting code is used, the text string will display on the current line until a newline is encountered. If no newline character is encountered before the end of the display screen line, the output is wrapped until a newline is found. DBCS SO/SI characters are added as necessary to make sure they are matched.

An example of :f format code usage is shown:

```

int main()
{
    char testc[]={"This is the first line.\nThis is the second line."
                  "\nThis is the third line."};

    int i;
    i = 1;
}

```

This program will result in the following screen output:

```

> EVAL *testc:s 100
*testc:s 100 =
    "This is the first line. This is the second line. This is the"
    "third line."
> EVAL *testc:f 100
*testc:f 100 =
    This is the first line.
    This is the second line.
    This is the third line.

```

Displaying Character Arrays

The following example shows the usage of the 'c' syntax to format an expression as characters. The array must be dereferenced by the '*' operator. If the * operator is not entered, the array will be displayed as a space pointer. If the dereferencing operator is used, but the 'c' is not appended to the expression, only the first array element is displayed. The default length of the display is 1.

1. While in a debug session, type DSPMODSRC, and press Enter. The Display Module Source display is shown.
2. Set a breakpoint at line 6.
3. Press F12(Resume) to leave the Display Module Source Display.
4. Call the program. The program stops at the breakpoint at line 6.
5. Type eval *array1: c 11 on the debug command line, and press Enter as shown:

```

Display Module Source
Program:  TEST3      Library:  DEBUG      Module:  MAIN
1  #include <string.h>
2  char array1 [11];
3  int i;
4  main(){
5      strcpy(array1,"0123456789");
6      i = 0;
7  }

Debug . . . eval *array1: c 11 _____ Bottom

F3=Exit program  F6=Add/Clear breakpoint  F10=Step  F11=Display variable
F12=Resume       F17=Watch variable       F18=Work with watch  F24=More keys

```

The following illustrates displaying 11 characters, including a null character. The null character appears as a blank.

```

Display Module Source
Program:  TEST3      Library:  DEBUG      Module:  MAIN
1  #include <string.h>
2  char array1 [11];
3  int i;
4  main(){
5      strcpy(array1,"0123456789");
6      i = 0;
7  }

Debug . . . _____ Bottom

F3=Exit program  F6=Add/Clear breakpoint  F10=Step  F11=Display variable
F12=Resume       F17=Watch variable       F18=Work with watch  F24=More keys
*array1: c 11 = '0123456789 ' ...

```

Using F11 to Display Variables

The easiest way to display data or an expression is to use F11 (Display variable) on the Display Module Source display. Place your cursor on the variable that you want to display and press F11. The current value of the variable is shown on the message line at the bottom of the Display Module Source display.

In cases where you are evaluating structures, records, classes, arrays, pointers, enumerations, bit fields, unions or functions, the message returned when you press F11 (Display variable) may span several lines. Messages that span several lines are shown on the Evaluate Expression display to show the entire text of the message. Once you have finished viewing the message on the Evaluate Expression display, press Enter to return to the Display Module Source display.

The Evaluate Expression display also shows all the past debug commands that you entered and the results from these commands. You can search forward or backward on the Evaluate Expression display for a specified string, or text and retrieve or reissue debug commands.

Sample EVAL Commands for Pointers, Variables, and Bit Fields

Figure 39 shows the use of the EVAL command with pointers, variables, and bit fields. The pointers, variables, and bit fields are based on the source in “Sample Source for EVAL Commands” on page 126.

Pointers

```
// Display a pointer
>eval pc1
pc1 = SPP:0000C0260900107C

// Assign a value to a pointer
>eval pc2=pc1
pc2=pc1 = SPP:0000C0260900107C

// Dereference a pointer
>eval *pc1
*pc1 = 'C'

// Take the address of a pointer
>eval &pc1
&pc1 = SPP:0000C02609001040

// Build an expression with normal C precedence
>eval *&pc1
*&pc1 = SPP:0000C0260900107C

// Casting a pointer
>eval *(short *)pc1
*(short *)pc1 = -15616

// Treat an unqualified array as a pointer
>eval arr1
arr1 = SPP:0000C02609001070

// Apply the array type through dereferencing
// (character in this example)
>eval *arr1
*arr1 = 'A'

// Override the formatting of an expression that is an lvalue
>eval *arr1:s
*arr1:s = "ABC"

// Set a pointer to null by assigning 0
>eval pc1=0
pc1=0 = SYP:*NULL

// Evaluate a function pointer
>eval fncptr
fncptr = PRP:0000A0CD0004F010

// Use the arrow operator
>eval *pY->x.p
*pY->x.p = ' '
```

Simple Variables

```
// Perform logical operations
>eval i1==u1 || i1<u1
i1==u1 || i1<u1 = 0
```

Figure 39. Sample EVAL Commands for Pointers, Variables, and Bit Fields (Part 1 of 2)

```

// Unary operators occur in proper order
>eval i1++
i1++ = 100

// i1 is incremented after being used
>eval i1
i1 = 101

// i1 is incremented before being used
>eval ++i1
++i1 = 102

// Implicit conversion
>eval u1 = -10
u1 = -10 = 4294967286

// Implicit conversion
>eval (int)u1
(int)u1 = -10

Bit Fields
// Display an entire structure
>eval bits
bits.b1 = 1
bits.b4 = 2

// Work with a single member of a structure
>eval bits.b4 = bits.b1
bits.b4 = bits.b1 = 1

// Bit fields are fully supported
>eval bits.b1 << 2
bits.b1 << 2 = 4

// You can overflow bit fields, but no warning is generated
>eval bits.b1 = bits.b1 << 2
bits.b1 = bits.b1 << 2 = 4
>eval bits.b1
bits.b1 = 0

```

Figure 39. Sample EVAL Commands for Pointers, Variables, and Bit Fields (Part 2 of 2)

The examples below show the use of the EVAL command with structures, unions, and enumerations. The structures, unions, and enumerations are based on the source in “Sample Source for EVAL Commands” on page 126.

Note: For C++, the structures are simple structures, not Classes.

```

Structures and Unions
// Cast with typedefs
>eval (struct z *)&zz
(struct z *)&zz = SPP:0000C005AA0010D0

// Cast with tags
>eval *(c *)&zz
(*(c *)&zz).a = 1
(*(c *)&zz).b = SYP:*NULL

```

Figure 40. Sample EVAL Commands for C Structures, Unions and Enumerations (Part 1 of 2)

```

Structures and Unions// Assign union members
>eval u.x = -10
u.x = -10 = -10

// Display a union. The union is formatted for each definition
>eval u
u.y = 4294967286
u.x = -10
Enumerations
// Display both the enumeration and its value
>eval Color
Color = blue (2)
>eval Number
Number = three (2)

// Cast to a different enumeration
>eval (enum color)Number
(enum color)Number = blue (2)

// Assign by number
>eval Number = 1
Number = 1 = two (1)

// Assign by enumeration
>eval Number = three
Number = three = three (2)

// Use enums in an expression
>eval arr1[one]
arr1[one] = 'A'

```

Figure 40. Sample EVAL Commands for C Structures, Unions and Enumerations (Part 2 of 2)

EVAL Commands for System and Space Pointers

The example below shows the use of the EVAL command with system and space pointers. The system and space pointers are based on the source in “Sample Source for Displaying System and Space Pointers” on page 127.

```

System and Space Pointers
// System pointers are formatted
// :1934:QTEMP      :111111110
>eval pSYSptr
pSYSptr =
        SYP:QTEUSERSPC
        0011100
// Space pointers return 8 bytes that can be used in
// System Service Tools
>eval pBuffer
pBuffer = SPP:0000071ECD000200

```

Figure 41. Sample EVAL Commands for System and Space Pointers

You can use the EVAL command on C and C++ language features and constructs. The ILE source debugger can display a full class or structure but only with those fields defined in the derived class. You can display a base class in full by casting the derived class to the particular base class.

The example below shows the use of the EVAL command with C++ language constructs. The C++ language constructs are based on the source in “Sample

Source for Displaying C++ Constructs” on page 129. Additional C++ examples are provided in the source debugger online help.

```
// Follow the class hierarchy (specifying class D is optional)
> EVAL *(class D *)this
  (*(class D *)this).__vbp1B = SPP:C40F5E3D7F000490
  (*(class D *)this).__vbp1D = SPP:C40F5E3D7F000440
  (*(class D *)this).d = 4

// Follow the class hierarchy (without specifying class D)
> EVAL *(D *)this
  (*(D *)this).__vbp1B = SPP:C40F5E3D7F000490
  (*(D *)this).__vbp1D = SPP:C40F5E3D7F000440
  (*(D *)this).d = 4

// Look at a local variable
> EVAL VAR
  VAR = 1

// Look at a global variable
> EVAL ::VAR
  ::VAR = 2

// Look at a class member (specifying this-> is optional)
> EVAL this->f
  this->f = 6

// Look at a class member (without specifying this->)
> EVAL f
  f = 6

// Disambiguate variable ac
> EVAL A::ac
  A::ac = 12

// Scope operator with template
> EVAL E<int>::ac
  E<int>::ac = 12

// Cast with template:
> EVAL *(E<int> *)this
  (*(E<int> *)this).__vbp1B = SPP:C40F5E3D7F000490
  (*(E<int> *)this).__vbp1EXTi_ = SPP:C40F5E3D7F000400
  (*(E<int> *)this).e = 5

// Assign a value to a variable
> EVAL f=23
  f=23 = 23

// See all local variables in a single EVAL statement
> EVAL %LOCALVARS
  local = 828
  this = SPP:C40F5E3D7F000400
  VAR = 1
```

Figure 42. Sample EVAL Commands for C++ Expressions

Displaying a Template Class and a Template Function

To display a template class or a template function, type EVAL template-name on the debug command line. The variable *template-name* is the name of the template class or template function you want to display.

The example below shows the results of evaluating a template class. You must enter a template name that matches the demangled template name. Typedef names

are not valid because the typedef information is removed when the template name is mangled.

```
> EVAL XX<int>::a
XX<int>::= '1 '
> EVAL XX<inttype>::a
Identifier not found
1  template < class A >           //Code evaluated at line 8
2  class XX {                     //where a breakpoint was set
3      static A a;
4      static B b;
5  };
6  XX<int> x;
7  typedef int inttype;
8  int XX<int>::a =1;             //mangled name a__2XXXTi_
9  int XX<inttype>::b = 2;       //mangled name b__2XXXTi_
```

Figure 43. Using EVAL with a Class Template

The example below shows the results of evaluating a template function.

```
> EVAL XX<int,12>::sxa
XX<int,12>::sxa = '1 '
> EVAL xxobj.xca[0]
xxobj.xca[0] = '2 '
1  template < class A, int B>     //Code evaluated at lines 8 and 9
2  class XX {                     //where breakpoints were set
3      static A sxa;
4      char    xca[B];
5  public:
6      XX(void) { xca[0] = 2; }
7  };
8  XX<int,12> xxobj;
9  int XX<int,2*6>::sxa =1;
                                     //same as intXX<int,12>::sxa
                                     //mangled name sxa__2XXXTiSP12_
```

Figure 44. Using EVAL with a Function Template

Changing Optimization and Observability

Once a program is created, it may need to be changed to address problems or changing user requirements. You may, for example, want to change the optimization level or observability of a module when you want to debug a program or when you are ready to put a program into production. For example, at higher levels of optimization, the values of variables may not be accurately displayed.

To circumvent this problem you can lower the optimization level of a module to display variables accurately as you debug a program, and then raise the level again afterwards to improve the program efficiency as you get the program ready for production.

Changing Optimization Levels

Optimizing an object means looking at the compiled code, determining what can be done to make the run time performance as fast as possible, and making the necessary changes. In general, the higher the optimizing request, the longer it takes

to create an object. At run-time the highly optimized program or service program should run faster than the corresponding non-optimized program or service program.

Example

This example shows you how to change the optimization level of module T1520IC4 from *FULL to *NONE to allow variables to be displayed and changed when the program is in debug mode. Once debug is complete, you can change the optimization level back to *FULL for improved run-time performance.

1. Type WRKMOD MODULE(T1520IC1) and press Enter. The Work with Modules display is shown.
2. Select option 5 (Display) to see the attribute values that need to be changed. The Display Module Information display is shown:

```

                                Display Module Information
Module . . . . . : T1520IC1
Library . . . . . : MYLIB
Detail . . . . . : *BASIC
Module attribute . . . . . : CLE
Module information:
  Module creation date/time . . . . . : 93/09/93 12:00:00
  Source file . . . . . : QACSRC
    Library . . . . . : MYLIB
  Source member . . . . . : T1520IC1
  Source file change date/time . . . . . : 93/08/18 13:31:40
  Owner . . . . . : SMITH
  Coded character set identifier . . . . . : 65535
  Text description . . . . . :
  Creation data . . . . . : *YES
  Intermediate language data . . . . . : *NO
                                          More...

Press Enter to continue.
F3=Exit  F12=Cancel

```

Note: In the display shown above, the Creation data value is *YES. This means that the module can be translated again once the optimization level value is changed. If the value is *NO, you must compile the module again in order to change the optimization level.

3. Press the Roll Down key to see more information for the module as shown:

```

                                Display Module Information
Module . . . . . : T1520IC4
Library . . . . . : MYLIB
Detail . . . . . : *BASIC
Module attribute . . . . . : CLE
  Sort sequence table . . . . . : *HEX
  Language identifier . . . . . : *JOB RUN
  Optimization level . . . . . : *NONE
  Maximum optimization level . . . . . : *FULL
  Debug data . . . . . : *YES
  Compressed . . . . . : *NO
  Program entry procedure name . . . . . : _C_pep
  Number of parameters . . . . . : 0 255
  Module state . . . . . : *USER
  Module domain . . . . . : *SYSTEM
  Number of exported defined symbols . . . . . : 1
  Number of imported (unresolved) symbols . . . . . : 10
  Press Enter to continue.
                                          More...

F3=Exit  F12=Cancel

```


Check the Maximum Optimization Level value. It may already be at the level you desire. If the module has the creation data, and you want to change the optimization level, press F12 (Cancel). The Work with Modules display is shown.

4. Select option 2 (Change) for the module whose optimization level you want to change. The CHGMOD command prompt is shown.
5. Type over the value specified for the field *Optimize Module*. Changing the module to a lower level of optimization allows you to display, and possibly change the value of variables while debugging. The following command would appear in the job log after the Enter key is pressed for the Change Module command if *NONE was entered as the optimization level.

```
CHGMOD MODULE(MYLIB/T1520IC4) OPTIMIZE(*NONE)
```

6. Do steps 2 through 5 again for any additional modules you may want to change. Whether you are changing one module or several in the same ILE program, the program creation time is the same because all imports are resolved when the system encounters them.

Note: Imports can be left unresolved using the *UNRSLVREF parameter of the CRTPGM command.

7. Create the program again using the CRTPGM command.

Removing Module Observability

Module observability involves two kinds of data that can be stored with a module, and that allows the module to be changed without being compiled again. Once a module is compiled, only this data can be removed. But if this data is removed, its observability is also removed, and you must recompile the module to replace the data. The two types of data are:

- | | |
|--------------------|---|
| Create Data | Represented by the *CRTDTA value. This data is necessary to translate the code to machine instructions. The module must have this data before you can change the module optimization level. |
| Debug Data | Represented by the *DBGDTA value. This data is necessary to allow a module to be debugged. |

Removing all observability reduces the module to its minimum size (with compression). It is not possible to change the module in any way unless you compile the module again. To compile it again, you must have authorization to access the source code.

Using the CHGMOD command, you can remove either kind of data from the module, remove both types, or remove none.

Example

Use the following procedure to remove observability from an ILE C/C++ program:

1. Type WRKMOD and press Enter. The Work with Modules display is shown.
2. Select option 5 (Display) to see the attribute values that need to be changed. The Display Module Information display is shown.

Check the value of the field *Creation data*. If it is *YES, the Create Data exists, and can be removed. If this value is *NO, there is no Create Data to remove. The module cannot be translated again unless you re-create it.

3. Press the Roll Down key to see more information for the module. Check the value of the field *Debug Data*. If it is *YES, the module can be debugged. If it is *NO, the module cannot be debugged unless you compile it again, and include the debug data.
4. Select option 2 (Change) for the module whose observability you want to change. The CHGMOD command prompt is shown.
5. Type over the value specified for the *Remove Observable Info* prompt. The following command appears in the job log for the Change Module command after the Enter key is pressed.
CHGMOD MODULE(MYLIB/T1520IC4) RMV OBS(*ALL)
6. You can ensure that the module is created again by changing the value of the *Force Module Recreation* parameter to *YES. This parameter is not required if the optimization level is changed. A change in the optimization level results in module re-creation unless the Create Data has been removed. However, if you want the program to be translated again after removing the debug data, and not changing the optimization level, you must use the *Force Module Recreation* parameter.
7. Do steps 2 through 5 again for any additional modules you want to change. Whether you are changing one module or several in the same ILE program, the program creation time is the same because all imports are resolved when the system encounters them.

Note: Imports can be left unresolved using the *UNRSLVREF parameter of the CRTPGM command. Program creation time is the same.

8. Create the ILE program again by using the CRTPGM command.

Sample Source for EVAL Commands

The sample EVAL commands presented in Figure 39 on page 119 and Figure 40 on page 120 are based on the following source:

```
#include <iostream.h>
#include <pointer.h>

/** POINTERS **/
_SYSPTR pSys;           //System pointer
_SPCPTR pSpace;         //Space pointer
int (*fncptr)(void);    //Function pointer
char *pcl;              //Character pointer
char *pc2;              //Character pointer
int *pil;               //Integer pointer
char arr1[] = "ABC";    //Array

/** SIMPLE VARIABLES **/
int il;                 //Integer
unsigned ul;            //Unsigned Integer
char cl;                //Character
float fl;               //Float

/** STRUCTURES **/
struct {                //Bit fields
    int b1 : 1;
    int b4 : 4;
}bits;
struct x{                // Tagged structure
    int x;
    char *p;
};
struct y{                // Structure with
```

```

        int y;                                // structure member
        struct x x;
    };
    typedef struct z {                        // Structure typedef
        int z;
        char *p;
    } z;
    z zz;                                    // Structure using typedef
    z *pZZ;                                  // Same
    typedef struct c {                        // Structure typedef
        unsigned a;
        char *b;
    } c;
    c d;                                    // Structure using typedef

    /** UNIONS **/
    union u{                                // Union
        int x;
        unsigned y;
    };
    union u u;                              // Variable using union
    union u *pU;                            // Same

    /** ENUMERATIONS **/
    enum number {one, two, three};
    enum color {red,yellow,blue};
    enum number Number = one;
    enum color Color = blue;

    /** FUNCTION **/
    int ret100(void) { return 100;}
    main()
    {
        float decl;
        struct y y, *pY;
        bits.b1 = 1;
        bits.b4 = 2;
        i1 = ret100();
        c1 = 'C';
        f1 = 100e2;
        decl = 12.3;
        pcl = &c1;
        pil = &i1;
        d.a = 1;
        pZZ = &zz;
        pZZ->z=1;
        pY = &y;
        pY->x.p=(char*)&y;
        pU=&u;
        pU->x=255;
        Number=(number)Color;
        fncptr = &ret100;
        pY->x.x=1;                            // Set breakpoint here
    }

```

Sample Source for Displaying System and Space Pointers



The sample EVAL command for displaying system and space pointers presented in Figure 41 on page 121 is based on the following source:

```

#include <iostream.h>
#include <misspace.h>
#include <pointer.h>

```

```

#include <mispobj.h>
#include <except.h>
#include <lecond.h>
#include <leenv.h>
#include <qtedbgs.h>          // From qsysinc

// Link up the Create User Space API
#pragma linkage(CreateUserSpace,OS)
#pragma map(CreateUserSpace,"QUSCRTUS")
void CreateUserSpace(char[20],
                    char[10],
                    long int,
                    char,
                    char[10],
                    char[50],
                    char[10],
                    _TE_ERROR_CODE_T *
                    );

// Link up the Delete User Space API
#pragma linkage>DeleteUserSpace,OS)
#pragma map>DeleteUserSpace,"QUSDLTUS")
void DeleteUserSpace(char[20],
                    _TE_ERROR_CODE_T *
                    );

// Link up the Retrieve Pointer to User Space API
#pragma linkage(RetrievePointerToUserSpace,OS)
#pragma map(RetrievePointerToUserSpace,"QUSPTRUS")
void RetrievePointerToUserSpace(char[20],
                                char **,
                                _TE_ERROR_CODE_T *
                                );

int main (int argc, char *argv[])
{
    char *pBuffer;
    _SYSPTR pSYSptr;
    _TE_ERROR_CODE_T errorCode;
    errorCode.BytesProvided = 0;
    CreateUserSpace("QTEUSERSPCQTEMP      ",
                  "QTESSPC      ",
                  10,
                  0,
                  "*ALL      ",
                  "      ",
                  "*YES      ",
                  &errorCode
                  );

    //!! call RetrievePointerToUserSpace - Retrieve Pointer to User Space
    //!! (pass: Name and library of user space, pointer variable
    //!! return: nothing (pointer variable is left pointing to start
    //!! of user space)
    RetrievePointerToUserSpace("QTEUSERSPCQTEMP      ",
                              &pBuffer,
                              &errorCode);

    // convert the space pointer to a system pointer
    pSYSptr = _SETSPFP(pBuffer);
    cout << "Space pointer: " << pBuffer << endl;
    cout << "System pointer: " << pSYSptr << endl;
    return 0;}

```

Sample Source for Displaying C++ Constructs



The sample EVAL command for displaying C++ constructs presented in Figure 42 on page 122 is based on the following source:

```
// Program demonstrates the EVAL debug command
class A {
public:
    union {
        int a;
        int ua;
    };
    int ac;
    int amb;
    int not_amb;
};

class B {
public:
    int b;
};

class C {
public:
    int ac;
    static int c;
    int amb;
    int not_amb;
};

int C::c = 45;
template <class T> class E : public A, public virtual B {
public:
    T e;
};

class D : public C, public virtual B {
public:
    int d;
};

class outter {
public:
    static int static_i;
    class F : public E<int>, public D {
    public:
        int f;
        int not_amb;
        void funct();
    } inobj;
};

int outter :: static_i = 45;

int VAR = 2;

void outter::F::funct()
{
    int local;
    a=1;           //EVAL VAR : Is VAR in global scope
    b=2;
    c=3;
```

```

d=4;
e=5;
f=6;

local = 828;
int VAR;

VAR=1;
static_i=10;
A::ac=12;
C::ac=13;
not_amb=32;

not_amb=13;
// Stop here and show:
// EVAL VAR          : is VAR in local scope
// EVAL ::VAR         : is VAR in global scope
// EVAL %LOCALVARS    : see all local vars
// EVAL *this         : fields of derived class
// EVAL this->f        : show member f
// EVAL f             : in derived class
// EVAL a             : in base class
// EVAL b             : in Virtual Base class
// EVAL c             : static member
// EVAL static_i      : static var made visible
//                               : by middle-end
// EVAL au            : anonymous union members
// EVAL a=49          :
// EVAL au            :
// EVAL ac            : show ambiguous var
// EVAL A::ac         : disambig with scope op
// EVAL B::ac         : Scope op
// EVAL E<int>::ac     : Scope op
// EVAL this          : notice pointer values
// EVAL (E<int>*)this  : change
// EVAL (class D *)this : class is optional
// EVAL *(E<int> *)this : show fields
// EVAL *(D *) this   : show fields

void main()
{
    outter obj;
    int outter::F::*mptr = &outter::F::b;
    int i;
    int& r = i;
    obj.inobj.funct();
    i = 777;

    obj.static_i = 2;
    // Stop here
    // EVAL obj.inobj.*mptr : member ptr
    // EVAL obj.inobj.b
    // EVAL i
    // EVAL r
    // EVAL r=1
    // EVAL i
    // EVAL (A &) (obj.inobj) : reference cast
    // EVAL
}

```

ILE Source Debugger and ILE C Application Hints

This sample program includes data definitions to illustrate what can be done with the ILE Source Debugger and ILE C applications.

```

#include <stdio.h>
#include <decimal.h>
#include <pointer.h>
/** POINTERS **/
_SYSPTR pSys;          /* System pointer */
_SPCPTR pSpace;         /* Space pointer */
int (*fncptr)(void);    /* Function pointer */
char *pc1;              /* Character pointer*/
char *pc2;              /* Character pointer*/
int *pi1;               /* Integer pointer */
char arr1[] = "ABC";    /* Array */
/** SIMPLE VARIABLES **/
int i1;                 /* Integer */
unsigned u1;            /* Unsigned Integer */
char c1;                /* Character */
float f1;               /* Float */
_Decimal(3,1) dec1;     /* Decimal */
/** STRUCTURES **/
struct {                /* Bit fields */
    int b1 : 1;
    int b4 : 4;
} bits;
struct x{               /* Tagged structure */
    int x;
    char *p;
};
struct y{               /* Structure with */
    int y;              /* structure member */
    struct x x;
};
typedef struct z {      /* Structure typedef*/
    int z;
    char *p;
} z;
z zz;                  /* Structure using typedef */
z *pZZ;

/* Same */
typedef struct c {      /* Structure typedef */
    unsigned a;
    char *b;
} c;
c d;                   /* Structure using typedef */
/** UNIONS **/
union u{               /* Union */
    int x;
    unsigned y;
};
union u u;             /* Variable using union */
union u *pU;           /* Same */
/** ENUMERATIONS **/
enum number {one, two, three};
enum color {red,yellow,blue};
enum number number = one;
enum color color = blue;

```

Figure 45. Sample ILE Source Debugger and ILE C Application (Part 1 of 2)

```

/** FUNCTION **/
int ret100(void) { return 100;}
main(){
    struct y y, *pY;
    bits.b1 = 1;
    bits.b4 = 2;
    i1 = ret100();
    c1 = 'C';
    f1 = 100e2;
    dec1 = 12.3;
    pc1 = &c1;
    pi1 = &i1;
    d.a = 1;
    pZZ = &zz;
    pZZ->z=1;
    pY = &y;
    pY->x.p=(char*)&y;
    pU = &u;
    pU->x=255;
    number=color;
    fncptr = &ret100;
    pY->x.x=1;          /* Set breakpoint here */
}

```

Figure 45. Sample ILE Source Debugger and ILE C Application (Part 2 of 2)

This screen illustrates some examples of using pointers in debug expressions.

Evaluate Expression	
Previous debug expressions	
>eval pc1	
pc1 = SPP:C0260900107C0000	Displaying pointers
>eval pc2=pc1	
pc2=pc1 = SPP:C0260900107C0000	Assigning pointers
>eval *pc1	
*pc1 = 'C'	Dereferencing pointers
>eval &pc1	
&pc1 = SPP:C026090010400000	Taking an address
>eval *&pc1	
*&pc1 = SPP:C0260900107C0000	Can build expressions with normal C precedence
>eval *(short *)pc1	
*(short *)pc1 = -15616	Casting
Debug . . .	Bottom
F3=Exit F9=Retrieve F12=Cancel F18=Command entry F19=Left F20=Right	

This screen illustrates some additional pointer examples.

Evaluate Expression	
Previous debug expressions	
>eval arr1 arr1 = SPP:C026090010700000	Unqualified arrays are treated as pointers
>eval *arr1 *arr1 = 'A'	Dereferencing applies the array type (character in this example)
>eval *arr1:s *arr1:s = "ABC"	If the expression is an lvalue you can override the formatting
>eval pc1=0 pc1=0 = SYP:*NULL	Setting a pointer to null by assigning 0
>eval fncptr fncptr = PRP:A0CD0004F0100000	Function pointers
>eval *pY->x.p *pY->x.p = ' '	Using the arrow operator
Bottom	
Debug . . . _____	
F3=Exit F9=Retrieve F12=Cancel F18=Command entry F19=Left F20=Right	

This screen illustrates some examples of simple variables used in debug expressions.

Evaluate Expression	
Previous debug expressions	
>eval i1==u1 i1<u1 i1==u1 i1<u1 = 0	Logical operations
>eval i1++ i1++ = 100	Unary operators occur in proper order
>eval i1 i1 = 101	Increment has happened after i1 was used
>eval ++i1 ++i1 = 102	Increment has happened before i1 was used
>eval u1 = -10 u1 = -10 = 4294967286	Implicit conversions happen
>eval (int)u1 (int)u1 = -10	
>eval decl decl = 12.3	Decimal types are displayed but cannot be used in expressions
Bottom	
Debug . . . _____	
F3=Exit F9=Retrieve F12=Cancel F18=Command entry F19=Left F20=Right	

This screen illustrates using bit fields in debug expressions.

Evaluate Expression

Previous debug expressions

>eval bits

bits.b1 = 1

bits.b4 = 2

>eval bits.b4 = bits.b1

bits.b4 = bits.b1 = 1

>eval bits.b1 << 2

bits.b1 << 2 = 4

>eval bits.b1 = bits.b1 << 2

bits.b1 = bits.b1 << 2 = 4

>eval bits.b1

bits.b1 = 0

You can display an entire structure

You can work with a single member

Bit fields are fully supported

You can overflow, but no warning is generated

Debug . . .

Bottom

F3=Exit F9=Retrieve F12=Cancel F18=Command entry F19=Left F20=Right

This screen illustrates using structures and unions in debug expressions.

Evaluate Expression

Previous debug expressions

>eval (struct z *)&zz

(struct z *)&zz = SPP:C005AA0010D00000

>eval *(c *)&zz

*(c *)&zz.a = 1

*(c *)&zz.b = SYP:*NULL

You can cast with typedefs

You can cast with tags

>eval u.x = -10

u.x = -10 = -10

>eval u

u.y = 4294967286

u.x = -10

You can assign union members

You can display and the union will be formatted for each definition

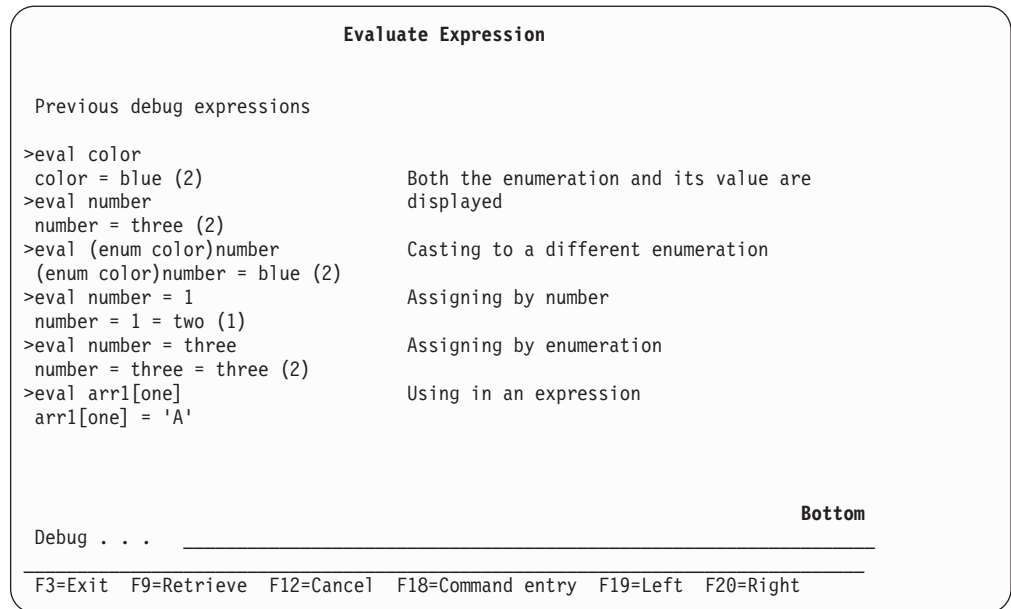
Debug . . .

Bottom

F3=Exit F9=Retrieve F12=Cancel F18=Command entry F19=Left F20=Right

This screen illustrates using enumerations in debug expressions.

134 ILE C/C++ Programmer's Guide



This sample program sets up system and space pointers for an example of how they can be displayed with the debugger.

```
#include <stdio.h>
#include <mispace.h>
#include <pointer.h>
#include <miscobj.h>
#include <except.h>
#include <lecond.h>
#include <leenv.h>
#include <qtedbgs.h>      /* From qsysinc */
/* Link up the Create User Space API */
#pragma linkage(CreateUserSpace,OS)
#pragma map(CreateUserSpace,"QUSCRTUS")
void CreateUserSpace(char[20],
                    char[10],
                    long int,
                    char,
                    char[10],
                    char[50],
                    char[10],
                    _TE_ERROR_CODE_T *
                    );
/* Link up the Delete User Space API */
#pragma linkage>DeleteUserSpace,OS)
#pragma map>DeleteUserSpace,"QUSDLTUS")
void DeleteUserSpace(char[20],
                    _TE_ERROR_CODE_T *
                    );
```

Figure 46. System and Space Pointers (Part 1 of 2)

```

/* Link up the Retrieve Pointer to User Space API */
#pragma linkage(RetrievePointerToUserSpace,OS)
#pragma map(RetrievePointerToUserSpace,"QUSPTRUS")
void RetrievePointerToUserSpace(char[20],
                                char **,
                                _TE_ERROR_CODE_T *)
;

int main (int argc, char *argv[])
{
    char *pBuffer;
    _SYSPTR pSYSptr;
    _TE_ERROR_CODE_T errorCode;
    errorCode.BytesProvided = 0;
    CreateUserSpace("QTEUSERSPCQTEMP      ",
                    "QTESSPC      ",
                    10,
                    0,
                    "*ALL      ",
                    "                                ",
                    "*YES      ",
                    &errorCode
                    );

    /*! call RetrievePointerToUserSpace - Retrieve Pointer to User Space */
    /*!! (pass: Name and library of user space, pointer variable */
    /*!! return: nothing (pointer variable is left pointing to start*/
    /*!!           of user space)                                */
    RetrievePointerToUserSpace("QTEUSERSPCQTEMP      ",
                               &pBuffer,
                               &errorCode);

    /* convert the space pointer to a system pointer */
    pSYSptr = _SETSPFP(pBuffer);

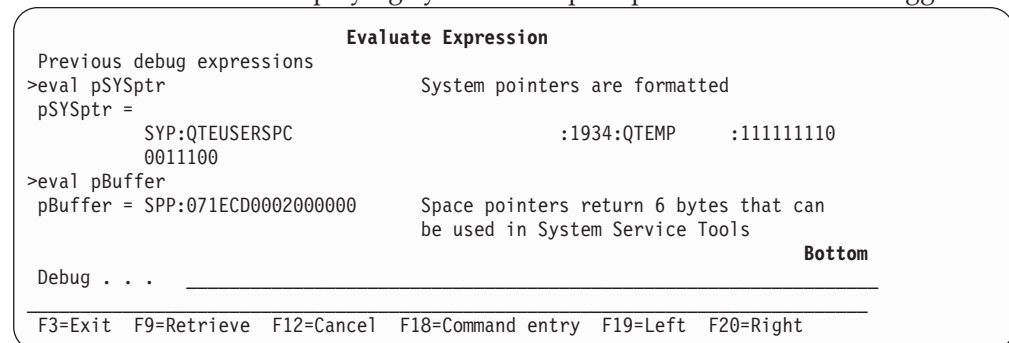
    printf("Space pointer: %p\n",pBuffer);
    printf("System pointer: %p\n",pSYSptr);

    return 0;
}

```

Figure 46. System and Space Pointers (Part 2 of 2)

This screen illustrates displaying system and space pointers with the debugger.



Debug Language Syntax

Limitations of the debug C expression grammar include:

1. **Type Casting:** Array and function designator type casts are prohibited.
2. **Function Calls:** Function calls cannot be used in debug expressions.
3. **Decimal Types:** Decimal types are supported for display only. They cannot be used in debug expressions.

Precedence of operators and type conversion of mixed types conforms to the C language.

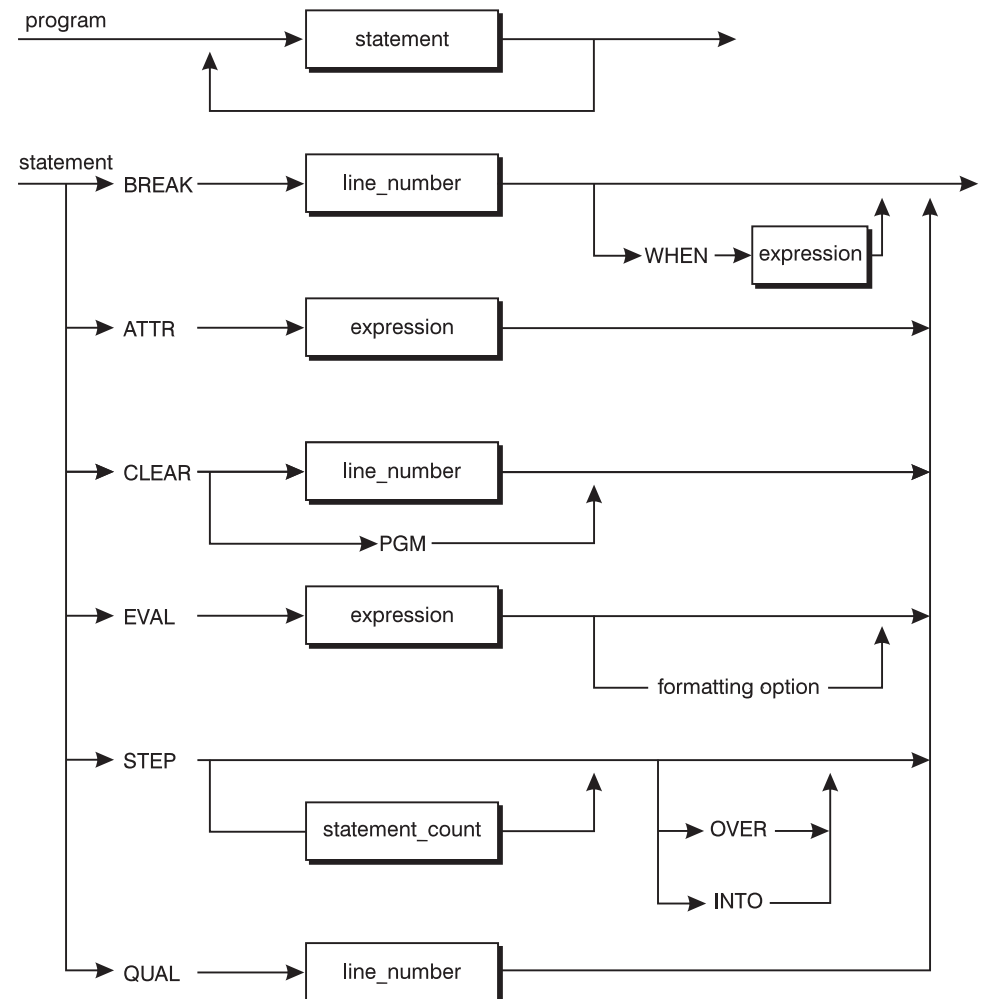


Figure 47. Debug Language Syntax (Program Flow and Program Data)

Part 4. Performing I/O Operations

This part describes:

- Store and operate on information in stream files
- Process stream files as true text or binary stream files
- Open text and binary stream files
- Perform I/O operations a record at a time
- Use open feedback and I/O feedback areas

Chapter 8. Using Stream and Record I/O Functions with iSeries Data Management Files

This chapter describes how to open, write, read, and update:

- Text stream files
- Binary stream files

The American National Standards Institute (ANSI) defines a C language **stream file** as a sequence of data that is read and written one character at a time. All I/O operations in ANSI C are stream operations.

On the iSeries Data Management system, all files are made up of records. All I/O operations at the operating system level are carried out a record at a time, using data management operations.

The ILE C/C++ run-time library allows your program to process stream files as text stream files or as binary stream files. Text stream files process one character at a time. Binary stream files process one character at a time or one record at a time. Since the iSeries Data Management system carries out I/O operations one record at a time, the ILE C/C++ library simulates stream file processing with OS/400 records. Although the ILE C/C++ library logically handles I/O one character at a time, the actual I/O that is performed by the operating system is done one record at a time.

Note: Since the iSeries Data Management system carries out I/O operations one record at a time, using system commands such as OPNQRYP together with stream I/O operations on the same file may cause positioning problems in the file your program is processing. Do not mix the use of ILE C/C++ extensions for record I/O and stream file functions on the same file as unpredictable results can occur. Avoid using system commands that logically work with records instead of characters in programs that contain stream I/O operations.

Record Files

The ILE C library provides a set of extensions to the ANSI C definition for I/O. This set of extensions, referred to as *record I/O*, allows your program to perform I/O operations one record at a time.

The ILE C record I/O functions work with all the file types that are supported on the iSeries system.

Each file that is opened with `_Ropen()` has an associated structure of type `_RFILE`. The `<recio.h>` header file defines this structure. Unpredictable results may occur if you attempt to change this structure.

Different open modes and keyword parameters apply to the different iSeries Data Management system file types. Chapter 10, "Using Externally Described Files in Your Programs" on page 185, Chapter 11, "Using Database Files and Distributed Data Management Files In Your Programs" on page 201, and Chapter 12, "Using Device Files in Your Programs" on page 219 provide information on each file type and how to open a record file using `_Ropen()`.

Note: There is no equivalent function provided by the C++ run-time library.

Stream Files

On the iSeries Data Management system, a stream is a continuous string of characters.

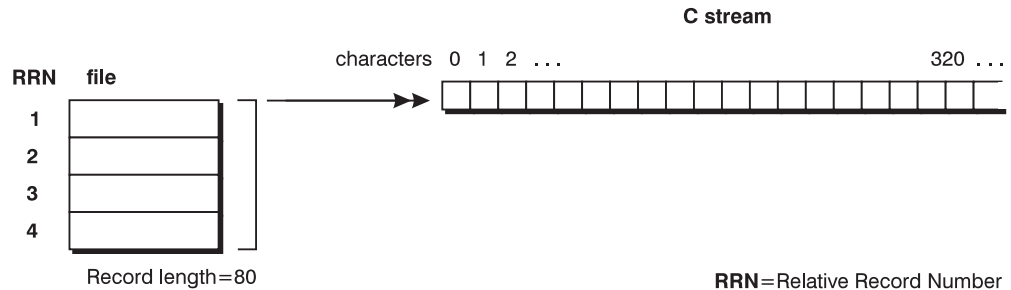


Figure 48. iSeries Data Management Records Mapping to an ILE C Stream File

The ILE C compiler allows your program to process stream files as text stream files or as binary stream files. See "Text Streams" on page 143 and "Binary Streams" on page 143.

Stream Buffering

Three buffering schemes are defined for ANSI standard C streams. They are:

- **Unbuffered** - characters are intended to appear from the source or at the destination, as soon as possible. The ILE C compiler does not support unbuffered streams.
- **Fully buffered** - characters are transmitted to and from a file one block at time, after the buffer is full. The ILE C compiler treats a block as the size of the system file's record.
- **Line buffered** - characters are transmitted to and from a file, as a block, when a new-line control character (`\n`) is encountered.

The ILE C compiler supports fully-buffered and line-buffered streams in the same manner, because a block and a line are equal to the record length of the opened file.

Note: The `setbuf()` and `setvbuf()` functions do not allow you to control buffering and buffer size when using the data management system.

Text Streams and Binary Streams

Each text stream file and each binary stream file is represented by a file control structure of type `FILE`. This structure is defined in the `<stdio.h>` header file. Unpredictable results may occur if you attempt to change the file control structure.

The format of `fopen()` is:

```
#include <stdio.h>
FILE *fopen(const char *filename, const char *mode);
```

The *mode* variable is a character string that consists of an open mode which may be followed by keyword parameters. The open mode and keyword parameters must be separated by a comma or one or more blank characters.



Create an input, output, or input/output file stream and then link to a file. Use the `open()` member function of the file stream class to link a file stream with a file. The prototype of the `open()` member function is:

```
void ifstream::open(const char *filename, openmode mode=ios::in);  
void ofstream::open(const char *filename, openmode mode=ios::out|ios::trunc);  
void fstream::open(const char *filename, openmode mode);
```

Text Streams

A **text stream** is an ordered sequence of characters that are composed of lines. Each line consists of zero or more characters and ends with a new-line character. The iSeries Data Management system may add, alter, or delete some special characters during input or output. Therefore, there may not be a one-to-one correspondence between the characters written to a text stream and characters read from the same text stream. When a file is closed, an implicit new-line character is appended to the end of the file unless a new-line character is already specified.

Data read from a text stream compares as equal to data written to the text stream if:

- The data consists of printable characters, horizontal tab, vertical tab, new-line character, or form-feed control characters.
- No new-line character is immediately preceded by a space (blank) character.
- The last character in a stream is a new-line character.
- The lines that are written to a file do not exceed the record length of the file.

Binary Streams

A **binary stream** is a sequence of characters that has a one-to-one correspondence with the characters stored in the associated iSeries Data Management system file. Character translation is not performed on binary streams. When data is written to a binary stream, it is the same when it is read back later. New-line characters have no special significance in a binary stream. On the iSeries system, the length of a binary stream file is a multiple of the record length. When a file is closed, the last record in the file is padded with nulls (hexadecimal value 0x00) to the end of the record.

Similar to text streams, binary streams map to records in iSeries Data Management system files. They can be processed one character at a time or one record at a time.

Open Modes for Dynamically Created Stream Files

If you specify the mode when opening a file, the iSeries Data Management system automatically creates a file if the file you specified does not already exist. A physical database file is created if you are using binary mode, or a source physical file is created if you are using text mode. If the file exists, but the member does not, the iSeries system adds the member to the file.

If you do not specify a library name when you open the file, the database file is dynamically created in library QTEMP. If you do not specify a member name, a member is created with the same name as the file.

The length that is specified on the `lrecl` parameter of `fopen()` is used for the record length of the file that is created, with the following exceptions:

- If you do not specify a record length when you open a text file, then a source physical file with a record length of 266 is created.
- If you do not specify a record length when you open a binary or record file, then a physical file with a record length of 80 is created.
- If you specify a record length of zero (lrecl=0) when you open a text file, then a source physical file with a record length of 266 is created.
- If you specify a record length of zero (lrecl=0) when you open a binary file, then a physical file with a record length of 80 is created.
- If the lrecl parameter is not specified for program-described files, then the record length that is specified on the CRTPRPG, or CRTPRTF is used. This length has a default value of 132, and if specified must be a minimum of 1.

Note: To use the source entry utility (SEU) to edit source files, specify an lrecl value of 240 characters or less on `fopen()`.

Dynamic file creation for text stream files is the same as specifying:

```
CRTSRCPF FILE(filename) RCDLEN(recLn)
```

Dynamic file creation for binary stream files is the same as specifying:

```
CRTPF FILE(filename) RCDLEN(recLn)
```

stdin, stdout, and stderr

When a program starts, three text streams are defined:

- Standard input (stdin) reads input from the terminal
- Standard output (stdout) writes output to the terminal
- Standard error (stderr) writes diagnostic output to the terminal.

Streams stdin, stdout, and stderr are implicitly opened the first time they are used.

- Stream stdin is opened with `fopen("stdin", "r")`.
- Stream stdout is opened with `fopen("stdout", "w")`.
- Stream stderr is opened with `fopen("stderr", "w")`.

These streams are not real iSeries Data Management system files, but are simulated as files by the ILE C library routines. By default, they are directed to the terminal session.

The stdin, stdout, and stderr streams can be associated with other devices using the OS/400 override commands on the files STDIN, STDOUT, and STDERR respectively. If stdin, stdout, and stderr are used, and a file override is present on any of these streams prior to opening the stream, then the override takes effect, and the I/O operation may not go to the terminal.

If stdout or stderr are used in a non-interactive job, and if there are no file overrides for the stream, then the ILE C compiler overrides the stream to the printer file QPRINT. Output prints or spools for printing instead of displaying at your workstation.

If stdin is specified (or the default accepted) for an input file that is not part of an interactive job, then the QINLINE file is used. You cannot re-read a file with QINLINE specified, because the database reader will treat it as an unnamed file, and therefore it cannot be read twice. You can avoid this by issuing an override. If

you are reading characters from `stdin`, F4 triggers the run time to end any pending input and to set the EOF indicator on. F3 is the same as calling `exit()` from your ILE C program.

If `stdin` is specified in batch and has no overrides associated with it, then `QINLINE` will be used. If `stdin` has overrides associated with it, then the override is used instead of `QINLINE`.

Note: You can use `freopen()` to reopen text streams. The `stdout` and `stderr` streams can be reopened for printer and database files. The `stdin` stream can be overridden only with database files.

Session Manager

ILE C stream I/O functions that output information to the display are defined through the Dynamic Screen Manager (DSM) session manager APIs. You can obtain the session handle for the C session and then use the DSM APIs to manipulate that session. The session handle is supplied through the `_C_Get_Ssn_Handle()` in `<stdio.h>`. For example, you can write a simple C program to clear the C session using the DSM `QsnClrScl` API, as shown in the following example:

```
#include <stdio.h>
#include "qsnapi.h"
void main (void)
{
    QsnClrScl(_C_Get_Ssn_Handle(), '0', NULL);
}
```

You can use the DSM APIs to perform any operation that is valid with a session handle, which includes the window services APIs and many of the low-level services also. You can display the session using a combination of the `QsnStrWin`, `QsnDspSsnBot`, and `QsnReadSsnDta` APIs, but it is simpler in this case to simply write a program that contains a `getc()`. As another example, you can use the `QsnRtvWinD` and `QsnChgWin` APIs to change the C session from the default full-screen window to a smaller window.

iSeries System Files

An ILE C stream file or record file is the same as an iSeries Data Management system file. System files are also called **file objects**. Each iSeries Data Management system file or file object is differentiated and categorized by information that is stored within it. Each file has its own set of unique characteristics, which determine how the file can be used and what capabilities it provides. This information is called the **file description**.

The file description also contains the file's characteristics, details on how the data associated with the file is organized into records, and how the fields are organized within these records. Whenever a file is processed, the operating system uses the file description. Data is created and accessed on the system through file objects.

The iSeries Data Management system files are listed:

- Database files store data on the iSeries Data Management system.
- Device files provide access to externally attached devices such as: displays, printers, tapes, and diskettes.

- 146 ILE C/C++ Programmer's Guide

Incorrect Character Hexadecimal Representation

(0x4D
*	0x5C
)	0x5D
/	0x6F
?	0x6F
'	0x7D
"	0x7F
(blank)	0x40

Note: "() / " can be used in quoted file names.

Opening Text Stream Files

To open an iSeries system file as a text stream file, use `fopen()` with one of the following modes:

- `r`
- `w`
- `a`
- `r+`
- `w+`
- `a+`

Notes:

1. The number of files that can be simultaneously opened by `fopen()` depends on the amount of the system storage available.
2. The `fopen()` function open modes also apply to the `freopen()` function.
3. If the text stream file contains deleted records, the deleted records are skipped by the text stream I/O functions.

The valid keyword parameters are:

- `lrecl`
- `ccsid`
- `recfm` (F, FA, and FB only)

If you specify a mode or keyword parameter that is not valid on `fopen()`, `errno` is set to `EBADMODE`, and `NULL` is returned.



To open an iSeries system file as a text stream file, use the `open()` member function with the following modes:

- `ios::app`
- `ios::ate`
- `ios::in`
- `ios::out`
- `ios::trunc`

Example

The following example illustrates how to open a text stream file. Library MYLIB must exist. The file TEST is created for you if it does not exist. The mode "w+" indicates that if MBR does not exist, it is created for update. If it does exist, it is cleared.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    FILE *fp;
    /* Open a text stream file. */
    /* Check to see if it opened successfully */
    if (( fp = fopen ( "MYLIB/TEST(MBR)", "w+" ) ) == NULL )
    {
        printf ( "Cannot open MYLIB/TEST(MBR)\n" );
        exit ( 1 );
    }

    printf ( "Opened the file successfully\n" );

    /* Perform some I/O operations. */

    fclose ( fp );
}
```

Figure 49. ILE C Source to Open an ILE C Text Stream File

Writing, reading, and updating can be performed on text stream files that are opened for processing.

Writing Text Stream Files

During a write operation, a new-line character in the buffer causes the remainder of the record written to the text stream file to be padded with blank characters (hexadecimal value 0x40). The new-line character itself is discarded.

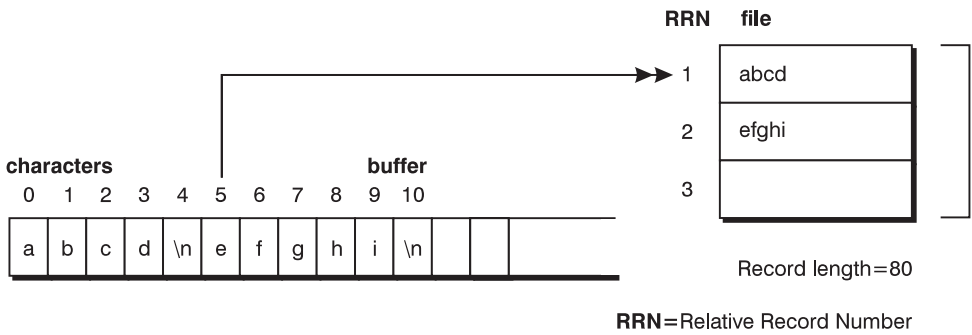


Figure 50. Writing to a Text Stream File

If the number of characters being written in the buffer exceeds the record length of the file, the data written to the file is truncated, and errno is set to ETRUNC.

Example

The following example illustrates how to write to a text stream file.


```

#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    char buf[12] = "abcd\nefghi\n";
    FILE *fp;
    /* Open a text file for writing.    */
    if (( fp = fopen ( "MYLIB/TEST(MBR)", "w" ) ) == NULL )
    {
        printf ( "Cannot open file\n" );
        exit ( 1 );
    }
    /* Write characters to the file.    */

    fputs ( buf, fp );

    /* Close the text file.            */

    fclose ( fp );
}

```

Figure 51. ILE C Source to Write Characters to a Text Stream File

Reading Text Stream Files

During a read operation from a text stream file, all the trailing blank characters (hexadecimal value 0x40) in the record that are read from the file into a buffer are ignored. A new-line character is inserted after the last non-blank.

Example

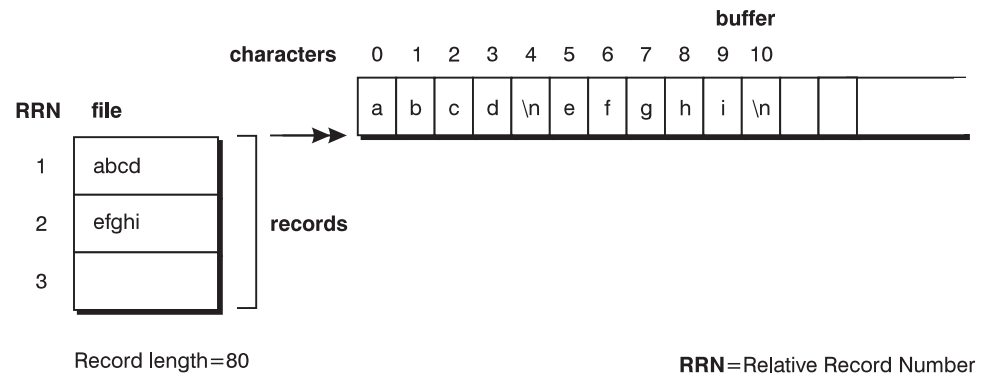


Figure 52. Reading from a Text Stream File

The following example illustrates how to read from a text stream file.

```

#include <stdio.h>
#include <stdlib.h>
void main(void)
{
    char buf[12];
    char *result;
    FILE *fp;
    /* Open an existing text file for reading. */
    if (( fp = fopen ( "MYLIB/TEST(MBR)", "r" ) ) == NULL )
    {
        printf ( "Cannot open file\n" );
        exit ( 1 );
    }
    /* Read characters into the buffer. */

    result = fgets ( buf, sizeof(buf), fp );
    printf("%10s", result);
    result = fgets ( buf+5, sizeof(buf), fp );
    printf("%10s", result);

    fclose ( fp );
}

```

Figure 53. ILE C Source to Read Characters from a Text Stream File

Updating Text Stream Files

During an update operation to a text stream file, if the number of characters being written to the file exceeds the record length of the file, trailing characters in the record are truncated and `errno` is set to `ETRUNC`.

If the data being written to the text stream file is shorter than the record length being updated, and the last character of the data being written is a new-line character, then the record is updated and the remainder of the record is filled with blank characters. If the last character of the data being written is not a new-line character, the record is updated and the remainder of the record remains unchanged.

Opening Binary Stream Files (one character at a time)

To open an iSeries Data Management system file as a binary stream file for character-at-a-time processing, use `fopen()` with any of the following modes:

- `rb`
- `wb`
- `ab`
- `r+b` or `rb+`
- `w+b` or `wb+`
- `a+b` or `ab+`

Notes:

1. The number of files that can be simultaneously opened by `fopen()` depends on the size of the system storage available.
2. The `fopen()` function open modes also apply to the `freopen()` function.
3. If the binary stream file contains deleted records, the deleted records are skipped by the binary stream I/O functions.

The valid keyword parameters are:

- blksize
- recfm
- commit
- arrseq
- lrecl
- type
- ccsid
- indicators

If you specify the type parameter the value must be memory for binary stream character-at-a-time processing.

Note: The memory parameter identifies this file as a memory file that is accessible only from C programs. This parameter is the default and is ignored.

If you specify a mode or keyword parameter that is not valid on `fopen()`, `errno` is set to `EBADMODE`.

Example

The following example illustrates how to open a binary stream file.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    FILE *fp;
    /* Open an existing binary file. */
    if (( fp = fopen ( "MYLIB/TEST(MBR)", "wb+" ) ) == NULL )
    {
        printf ( "Cannot open file\n" );
        exit ( 1 );
    }
    printf ("Opened the file successfully\n");

    /* Perform some I/O operations. */

    fprintf (fp, "Hello, world");

    fclose ( fp );
}
```

Figure 54. ILE C Source to Open a Binary Stream File

Writing, reading, and updating can be performed on binary stream files opened for character-at-a-time processing.

► C++

To open an iSeries Data Management system file as a binary stream file for character-at-a-time processing, use the `OPEN()` member function with `ios::binary` as well as any of the following modes:

- `ios::app`
- `ios::ate`
- `ios::in`
- `ios::out`
- `ios::trunc`

Writing Binary Stream Files (one character at a time)

If you write data to a binary stream processed one character at a time, and the size of the data is greater than the current record length, then the excess data is written to the current record up to its record size and the remaining data is written to the next record in the file.

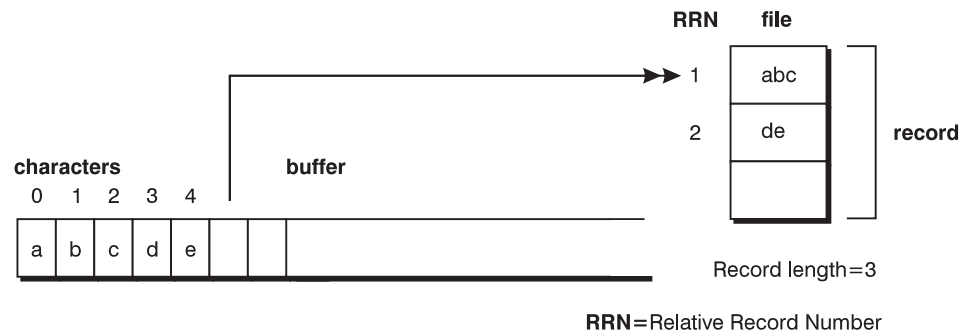


Figure 55. Writing to a Binary Stream File One Character at a Time

Example

The following example illustrates how to write to a binary stream by character.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    FILE *fp;
    char buf[5] = {'a', 'b', 'c', 'd', 'e'};
    /* Open an existing binary file for writing. */
    if (( fp = fopen ( "MYLIB/TEST(MBR)", "wb" ) ) == NULL )
    {
        printf ( "Cannot open file\n" );
        exit ( 1 );
    }
    /* Write 5 characters from the buffer to the file. */

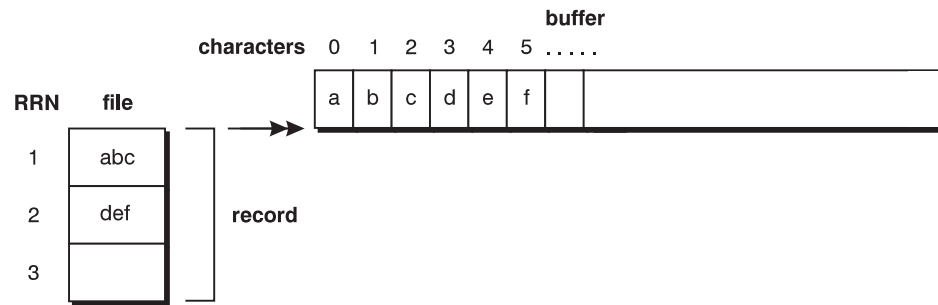
    fwrite ( buf, 1, sizeof(buf), fp );

    fclose ( fp );
}
```

Figure 56. ILE C Source to Write Characters to a Binary Stream File

Reading Binary Stream Files (one character at a time)

During a read operation from a binary stream that is processed a character at a time, if the length of the data being read is greater than the record length of the file, then data is read from the next record in the file.



Record length=3

RRN=Relative Record Number

Figure 57. Reading from a Binary Stream File One Character at a Time

Example

The following illustrates how to read from a binary stream file by character.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    FILE *fp;
    char buf[6];
    /* Open an existing binary file for reading. */
    if (( fp = fopen ( "MYLIB/TEST(MBR)", "rb" ) ) == NULL )
    {
        printf ( "Cannot open file\n" );
        exit ( 1 );
    }
    /* Read characters from the file to the buffer. */

    fread ( buf, 1, sizeof(buf), fp );
    printf ( "%6s\n", buf );

    fclose ( fp );
}
```

Figure 58. ILE C Source to Read Characters from a Binary Stream File

Updating Binary Stream Files (one character at a time)

If the amount of data being updated exceeds the current record length, then the excess data updates the next record. If the current record is the last record in the file, a new record is created.

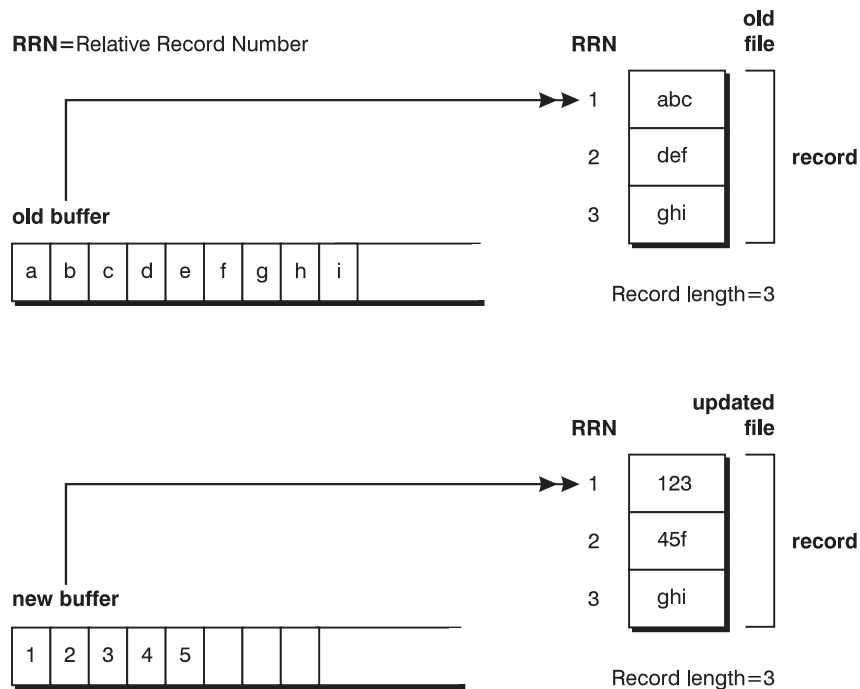


Figure 59. Updating a Binary Stream File with Data Longer than Record Length

Example

The following example illustrates updating a binary stream file with data that is longer than the record length.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    FILE *fp;
    char buf[5] = "12345";
    /* Open an existing binary file for updating. */
    if ( ( fp = fopen ( "QTEMP/TEST(MBR)", "rb+" ) ) == NULL )
    {
        printf ( "Cannot open file\n" );
        exit ( 1 );
    }
    /* Write 5 characters from the buffer to the file. */

    fwrite ( buf, 1, sizeof(buf), fp );

    fclose ( fp );
}
```

Figure 60. ILE C Source to Update a Binary Stream File with Data Longer than the Record Length

If the amount of data being updated is shorter than the current record length, then the record is partially updated and the remainder is unchanged.

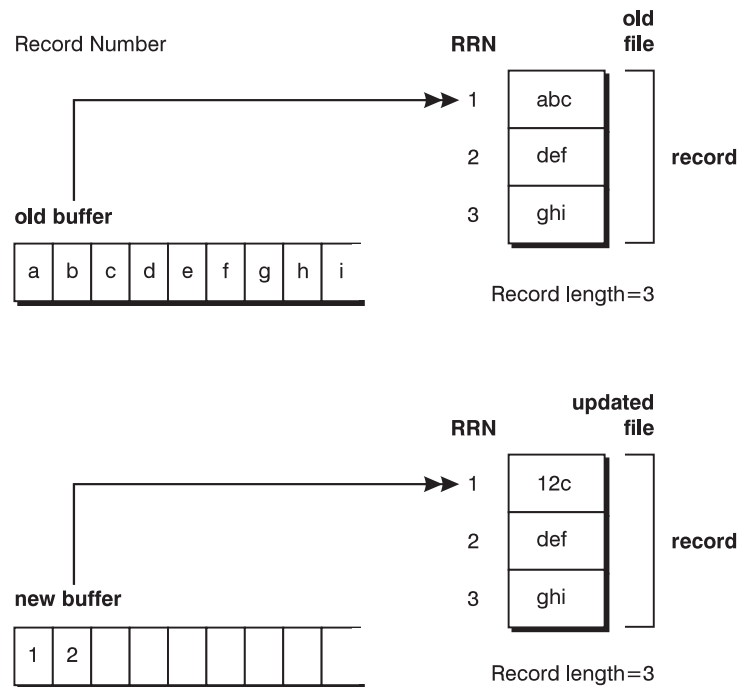


Figure 61. Updating a Binary Stream File With Data Shorter than Record Length

Example

The following example illustrates updating a binary stream file with data that is shorter than the record length.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    FILE *fp;
    char buf[2] = "12";
    /* Open an existing binary file for updating. */
    if (( fp = fopen ( "QTEMP/TEST(MBR)", "rb+" ) ) == NULL )
    {
        printf ( "Cannot open file\n" );
        exit ( 1 );
    }
    /* Write 2 characters from the buffer to the file. */

    fwrite ( buf, 1, sizeof(buf), fp );

    fclose ( fp );
}
```

Figure 62. ILE C Source to Update a Binary Stream File with Data Shorter than the Record Length

Opening Binary Stream Files (one record at a time)

To open an iSeries Data Management system file as a binary stream file for record-at-a-time processing, use `fopen()` with any of the following modes:

- `rb`
- `wb`
- `ab`
- `r+b` or `rb+`
- `w+b` or `wb+`
- `a+b` or `ab+`

Notes:

1. The number of files that can be simultaneously opened by `fopen()` depends on the size of the system storage available.
2. The `fopen()` open modes also apply to `freopen()`.
3. If the binary stream file contains deleted records, the deleted records are skipped by the binary stream I/O functions.
4. The file must be opened with the type set to record.

The valid keyword parameters are:

- `blksize`
- `recfm`
- `commit`
- `arrseq`
- `lrecl`
- `type`
- `ccsid`
- `indicators`

If you specify a mode or keyword parameter that is not valid on `fopen()` function, `errno` is set to `EBADMODE`.

Only `fread()` and `fwrite()` can be used for binary stream files opened for record-at-a-time processing.



To open an iSeries Data Management system file as a binary stream file for record-at-a-time processing, use the `OPEN()` member function with `ios::binary` as well as any of the following modes:

- `ios::app`
- `ios::ate`
- `ios::in`
- `ios::out`
- `ios::trunc`

Writing Binary Stream Files (one record at a time)

If you write data to a binary stream processed one record at a time, and the product of size and count (parameters of `fwrite()`) is greater than the record length, then only the data that fits in the current record is written and `errno` is set to `ETRUNC`.

If the product of size and count is less than the actual record length, the current record is padded with blank characters and `errno` is set to `EPAD`.

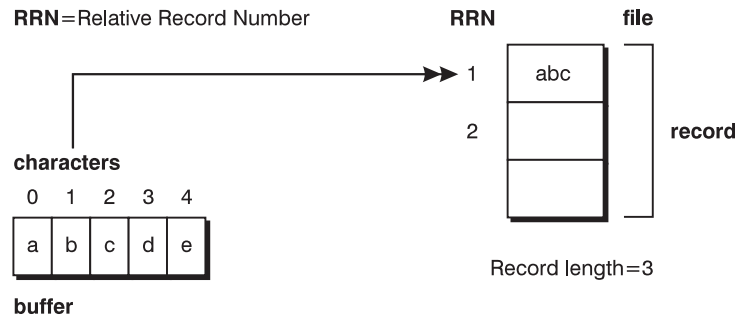


Figure 63. Writing to a Binary Stream File One Record at a Time

Only `fwrite()` is valid for writing to binary stream files opened for record-at-a-time processing. All other output and positioning functions fail, and `errno` is set to `ERECIO`.

Example

The following example illustrates how to write to a binary stream file by record.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    FILE *fp;
    char buf[5] = {'a', 'b', 'c', 'd', 'e'};
    /* Open an existing binary file for writing. */
    if ((fp = fopen ( "MYLIB/TEST(MBR)", "wb,type=record,lrec1=3" ))==NULL)
    {
        printf ( "Cannot open file\n" );
        exit ( 1 );
    }
    /* Write 3 characters from the buffer to the file. */

    fwrite ( buf, 1, sizeof(buf), fp );

    fclose ( fp );
}
```

Figure 64. ILE C Source to Write to a Binary Stream File by Record

Reading Binary Stream Files (one record at a time)

If you read data from a binary stream processed one record at a time, and the product of size and count (parameters of `fread()`) is greater than the record length, then only the data in the current record is read into the buffer. The `fread()` function returns a value indicating that there is less data in the buffer than was specified.

If the product of size and count is less than the actual record length, `errno` is set to `ETRUNC` to indicate that there is data in the record that was not copied into the buffer.

This figure illustrates how only the current record is read into the buffer, when the product of size and count is greater than the record length.

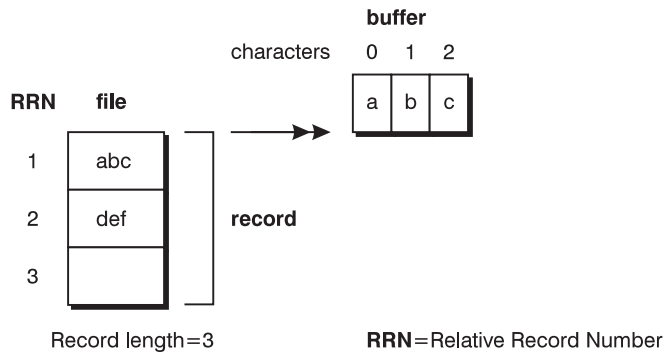


Figure 65. Reading From a Binary Stream File a Record at a Time

Only `fread()` function is valid for reading binary stream files opened for record-at-a-time processing. All other input and positioning functions fail, and `errno` is set to `ERECIO`.

Example

The following example illustrates how to read a binary stream a record at a time.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    FILE *fp;
    char buf[6];
    /* Open an existing binary file for reading a record at a time. */
    if (( fp = fopen ( "MYLIB/TEST(MBR)", "rb, type=record" ) ) == NULL )
    {
        printf ( "Cannot open file\n" );
        exit ( 1 );
    }
    /* Read characters from the file to the buffer. */
    fread ( buf, sizeof(buf), 1, fp );
    printf ( "%6s\n", buf );

    fclose ( fp );
}
```

Figure 66. ILE C Source to Read from a Binary Stream File by Record

Open Feedback Area

The open feedback area is part of the open data path that contains information about the open file that is associated with that open data path. You can assign a pointer to this information by using the `_Ropnfbk()` function. The structure that maps to the open feedback area can be that is found in the `xxfdbk.h` header file.

I/O Feedback Area

The I/O feedback area is a part of the open data path for the file that is updated after each successful non-blocked I/O operation. If record blocking is taking place, the I/O feedback is updated after each block of records is transferred between your program and the Data Management system.

The I/O feedback consists of two parts: one part that is common to all file types, and one part that is specific to the type of file.

To assign a pointer to the common part of the I/O feedback area, use the `_Riobufk()` function. To assign a pointer to the part of the I/O feedback area that is specific to the type of file, add the offset contained in the `file_dep_fb_offset` field of the common part to a pointer to the common part.

Note: The offset is in bytes, so you need to cast the pointer (`char *`) to the common part to a pointer to character when performing the pointer arithmetic. The structures that map to the I/O feedback areas are that are contained in the `xxfdbk.h` header file.

Chapter 9. Using ILE C/C++ Stream Functions with the iSeries Integrated File System

This chapter describes how to open, write, read, and update text and binary stream files through the iSeries Integrated File System.

The integrated file system provides a common interface to store and operate on information in stream files. Examples of stream files are PC files, files in UNIX[®] systems, LAN server files, iSeries files, and folders.

Note: The ILE C/C++ integrated file system enabled stream I/O functions are defined through the integrated file system. You need to be familiar with the integrated file system to use the ILE C/C++ stream I/O function. There are 7 file systems that make up the integrated file system. Depending on your application and environment, you may use several of the file systems. If you have existing applications that use iSeries system files, you need to understand the limitations of the new QSYS.LIB file system. If you have new applications, you can use the other file systems which do not have the QSYS.LIB file handling restrictions. See "The Integrated File System" section for information on each file system.

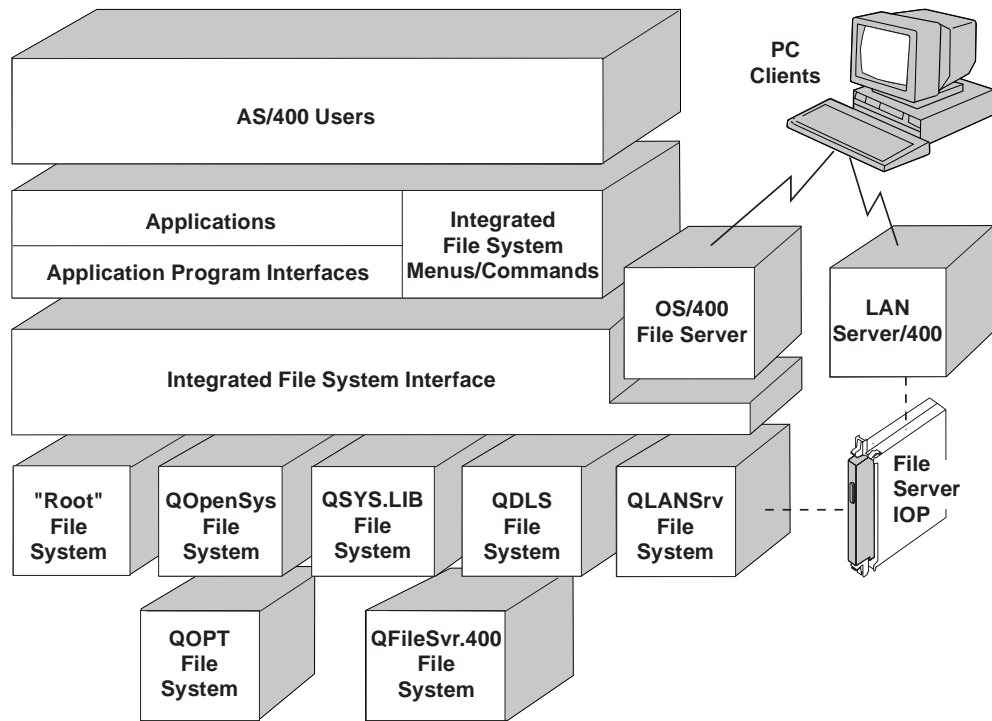
The Integrated File System

A **file system** provides the support that allows applications to access specific segments of storage that are organized as logical units. These logical units are files, directories, libraries, and objects. There are seven file systems in the Integrated File System:

- "root" (/)
- Open systems (QOpenSys)
- Library (QSYS.LIB)
- Document library services (QDLS)
- LAN Server/400 (QLANSrv)
- Optical support (QOPT)
- File server (QFileSvr.400)

Figure 67 on page 162 illustrates these file systems.

Users and application programs can interact with any of the file systems through a common **Integrated File System interface**. This interface is optimized for input/output of stream data, in contrast to the record input/output that is provided through the data management interfaces. The common integrated file system interface includes a set of user interfaces (commands, menus, and displays) and application program interfaces (APIs).



RV3N501-0

Figure 67. The Integrated File System Interface

Root File System

The "root" (/) file system can be accessed only through the integrated file system interface. You work with the "root" (/) file system using integrated file system commands, user displays, or ANSI stream I/O functions and system APIs.

This file system is designed to take full advantage of the stream file support and hierarchical directory structure of the integrated file system. It has the characteristics of the DOS and OS/2 file systems.

Path Names

This file system preserves the same uppercase and lowercase form in which object names are entered, but no distinction is made between uppercase and lowercase when the system searches for names.

- Path names have the following form:
Directory/Directory/ . . . /Object
- Each component of the path name can be up to 255 characters long. The path can be up to 16 megabytes.
- There is no limit on the depth of the directory hierarchy other than program and space limits
- The characters in names are converted to Universal Character Set 2 (UCS2) Level 1 form when the names are stored.

Open Systems File System

The open systems (QOpenSys) file system is designed to be compatible with UNIX-based open system standards, such as POSIX and XPG. Like the "root" (/)

file system, it takes advantage of the stream file and directory support provided by the integrated file system. In addition, it supports case-sensitive object names.

QOpenSys can be accessed only through the integrated file system interface. You work with QOpenSys using integrated file system commands, user displays, or ANSI stream I/O functions and system APIs.

Path Names

Unlike the QSYS.LIB, QDLS, QLANSrv, and "root" (/) file systems, the QOpenSys file system distinguishes between uppercase or lowercase characters when searching object names.

The path names, link support, commands, displays and ANSI stream I/O functions and system APIs are the same as defined under the "root" (/) file system.

Library File System

The library (QSYS.LIB) file system supports the iSeries library structure. It provides access to database files and all of the other iSeries object types that are managed by the library support.

The QSYS.LIB file system maps to the iSeries file system. For example, the path /qsys.lib/qsysinc.lib/h.file/stdio.mbr refers to the data management file STDIO, in the file H, in library QSYSINC, within the root library QSYS.

File Handling Restrictions

There are some limitations in using the integrated file system facilities:

- Logical files are not supported.
- The only types of physical files that are supported are program-described files that contain a single field, and source physical files that contain a single text field.
- Byte-range locking is not supported.
- If any job has a database file member open, only one job is given write access to that file at any time; other jobs are allowed only read access.

Path Names

In general, the QSYS.LIB file system does not distinguish between uppercase and lowercase names of objects. A search for object names achieves the same result regardless of whether characters in the names are uppercase or lowercase.

However, if the name is enclosed in quotation marks, the case of each character in the name is preserved. The search is sensitive to the case of characters in quoted names.

Each component of the path name must contain the object name followed by the object type. For example:

```
/QSYS.LIB/QGPL.LIB/PRT1.OUTQ  
/QSYS.LIB/PAYROLL.LIB/PAY.FILE/TAX.MBR
```

The object name and object type are separated by a period (.). Objects in a library can have the same name if they are different object types, so the object type must be specified to uniquely identify the object.

The object name in each component can be up to 10 characters long, and the object type can be up to 6 characters long.

The directory hierarchy within QSYS.LIB can be either two or three levels deep (two or three components in the path name), depending on the type of the object being accessed. If the object is a database file, the hierarchy can contain three levels (library, file, member); otherwise, there can be only two levels (library, object). The combination of the length of each component name and the number of directory levels determines the maximum length of the path name.

If "root" (/) and QSYS.LIB are included as the first two levels, the directory hierarchy for QSYS.LIB can be four or five levels deep.

The characters in names are converted to code page 37 when the names are stored. Quoted names are stored using the code page of the job.

Document Library Services File System

The document library services (QDLS) file system supports the folder objects. It provides access to documents and folders.

To work with the QDLS file system through the integrated file system interface, use the integrated file system commands, user displays, or ANSI stream I/O functions and system APIs.

All users working with objects in QDLS must be enrolled in the system distribution directory.

Path Names

QDLS does not distinguish between uppercase and lowercase in the names containing only the alphabetic characters a to z. A search for object names achieves the same result regardless of whether characters in the names are uppercase or lowercase.

Other characters are case sensitive and are used as is.

Each component of the path name can consist of just a name, such as:

/QDLS/FLR1/DOC1

or a name plus an extension, such as:

/QDLS/FLR1/DOC1.TXT

The name in each component can be up to 8 characters long, and the extension can be up to 3 characters long. The maximum length of the path name is 82 characters.

The directory hierarchy below /QDLS/ can be 32 levels deep.

The characters in names are converted to code page 500 when the names are stored. A name may be rejected if it cannot be converted to code page 500.

LAN Server/400 File System

The LAN Server/400 (QLANSrv) file system provides access to the same directories and files that are accessed through the LAN Server/400 licensed program. It allows users of the OS/400 file server and iSeries applications to use the same data as LAN Server/400 clients.

To work with the QLANSrv file system through the integrated file system interface, use the integrated file system commands, user displays, or ANSI stream I/O functions and system APIs.

Files and directories in the QLANSrv file system are stored and managed by a LAN server that is based on the OS/2 LAN server. This LAN server does not support the concept of a file or directory owner or owning group. File ownership cannot be changed using a command or an ANSI stream I/O function and system API. Access is controlled through **access control lists**. You can change these lists by using the WRKAUT and CHGAUT commands.

Path Names

The file system preserves the same uppercase and lowercase form in which object names are entered. No distinction is made between uppercase and lowercase when the system searches for names. A search for object names achieves the same result regardless of whether characters in the names are uppercase or lowercase.

- Path names have the following form:
Directory/Directory/ . . . /Object
- Each component of the path name can be up to 255 characters long.
- The directory hierarchy within QLANSrv can be 127 levels deep. If all components of a path are included as hierarchy levels, the directory hierarchy can be 132 levels deep.
- Names are stored in the code page that is defined for the File Server.

Optical Support File System

The QOPT file system can be accessed through the integrated file system interface. This is done using either the OS/400 file server or the integrated file system commands, user displays, and ANSI stream I/O functions, and system APIs.

Path Names

QOPT converts the lowercase English alphabetic characters a to z to uppercase when used in object names. Therefore, a search for object names that uses only those characters is not case-sensitive.

- The path name must begin with a slash (/) and contain no more than 294 characters. The path is made up of the file system name, the volume name, the directory and subdirectory names, and the file name. For example:
/QOPT/VOLUMENAME/DIRECTORYNAME/SUBDIRECTORYNAME/FILENAME
- The file system name, QOPT, is required.
- The volume name is required and can be up to 32 characters long.
- One or more directories or subdirectories can be included in the path name, but none are required. The total number of characters in all directory and subdirectory names, including the leading slash, cannot exceed 63 characters. Directory and file names allow any character except 0x00 through 0x3F, 0xFF, 0x80, lowercase-alphabetic characters, and the following characters:
 - Asterisk (*)
 - Hyphen (-)
 - Question mark (?)
 - Quotation mark (")
 - Greater than (>)
 - Less than (<)
- The file name is the last element in the path name. The file name length is limited by the directory name length in the path. The directory name and file name that are combined cannot exceed 256 characters, including the leading slash.

- The characters in names are converted to code page 500 within the QOPT file system. A name may be rejected if it cannot be converted to code page 500. Names are written to the optical media in the code page that is specified when the volume was initialized.

File Server File System

The QFileSvr.400 file system can be accessed through the integrated file system interface. This is done by using either the OS/400 file server or the integrated file system commands, user displays, and ANSI stream I/O functions and system APIs. In using the integrated file system interfaces, you should be aware of the following considerations and limitations.

Note: The characteristics of the QFileSvr.400 file system are determined by the characteristics of the file system that are being accessed on the target system.

Path Names

For a first-level directory, which actually represents the "root" (/) directory of the target system, the QFileSvr.400 file system preserves the same uppercase and lowercase form in which object names are entered. However, no distinction is made between uppercase and lowercase when QFileSvr.400 searches for names.

For all other directories, case-sensitivity is dependent on the specific file system being accessed. QFileSvr.400 preserves the same uppercase and lowercase form in which object names are entered when file requests are sent to the OS/400 file server.

- Path names have the following form:

```
/QFileSvr.400/RemoteLocationName/Directory/Directory . . . /Object
```

The first-level directory (that is, RemoteLocationName in the example shown above) represents both of the following:

- The name of the target system that will be used to establish a communications connection. The target system name can be either of the following:
 - A TCP/IP host name (for example, beowulf.newyork.corp.com)
 - An SNA LU 6.2 name (for example. appn.newyork).
- The "root" (/) directory of the target system

Therefore, when a first-level directory is created using an integrated file system interface, any specified attributes are ignored.

Note: First-level directories are not persistent across IPLs. That is, the first-level directories must be created again after each IPL.

- Each component of the path name can be up to 255 characters long. The absolute path name can be up to 16 megabytes long.

Note: The file system in which the object resides may restrict the component length and path name length to less than the maximum allowed by QFileSvr.400.

- There is no limit to the depth of the directory hierarchy, other than program and system limits, and any limits that are imposed by the file system being accessed.
- The characters in names are converted to UCS2 Level 1 form when the names are stored.

Enabling Integrated File System Stream I/O

You can enable ILE C/C++ stream I/O for files up to two gigabytes in size by specifying the *IFSIO option on the system interface keyword (SYSIFCOPT) on the Create Module or Create Bound Program command prompt. For example:

```
CRTCMOD MODULE(QTEMP/IFSIO) SRCFILE(QCLE/QACSRC) SYSIFCOPT(*IFSIO)
CRTBNDC PGM(QTEMP/IFSIO) SRCFILE(QCLE/QACSRC) SYSIFCOPT(*IFSIO)
```

Using Stream I/O with Large Files

The 64-bit version of the Integrated File System interface lets you use ILE C/C++ Stream I/O with files greater than two gigabytes in size. Use any of the methods listed below to enable this interface.

- Specify the *IFS64IO option with the SYSIFCOPT keyword on the Create Module or Create Bound Program command prompt. When this option is specified, the compiler defines the `__IFS64_IO__` macro, which in turn causes the `_LARGE_FILES` and `_LARGE_FILE_API` macros to be defined in the IBM-supplied header files. For example:

```
CRTCPMOD MODULE(QTEMP/IFSIO) SRCFILE(QCLE/QACSRC) SYSIFCOPT(*IFS64IO)
```

- Define the `_LARGE_FILES` macro in the program source. Alternately, specify `DEFINE('_LARGE_FILES')` on a Create Module or Create Bound Program command line. Integrated File System APIs and relevant data types are automatically mapped or redefined to their 64-bit Integrated File System counterparts. For example:

```
CRTCPMOD MODULE(QTEMP/IFSIO) SRCFILE(QCLE/QACSRC)
SYSIFCOPT(*IFSIO) DEFINE('_LARGE_FILES')
```

- Define the `_LARGE_FILE_API` macro in the program source. Alternately, specify `DEFINE('_LARGE_FILE_API')` on a Create Module or Create Bound Program command line. This makes 64-bit Integrated File System APIs and corresponding data types visible, but applications must explicitly specify which Integrated File System APIs (regular or 64-bit) to use. For example:

```
CRTCMOD MODULE(QTEMP/IFSIO) SRCFILE(QCLE/QACSRC)
SYSIFCOPT(*IFSIO) DEFINE('_LARGE_FILE_API')
```

Notes on Usage

- The `__IFS64_IO__`, `_LARGE_FILES`, and `_LARGE_FILE_API` macros are not mutually exclusive. For example, you might specify `SYSIFCOPT(*IFS64IO)` on the command line, and define either or both of the `_LARGE_FILES` and `_LARGE_FILE_API` macros in your program source.

Stream Files

The ILE C/C++ compiler allows your program to process stream files as true text or binary stream files (using the integrated file system enabled stream I/O) or as simulated text and binary stream files (using the default data management stream I/O).

When writing an application that uses stream files, for better performance, it is recommended that the integrated file system be used instead of the default C stream I/O which is mapped on top of the data management record I/O.

Stream Files Versus Database Files

To better understand stream files, it is useful to compare them with iSeries database files.

On the integrated file system, a stream is simply a continuous string of characters. A database file is record arranged; It has predefined subdivisions consisting of one or more fields that have specific characteristics, such as length and data type.

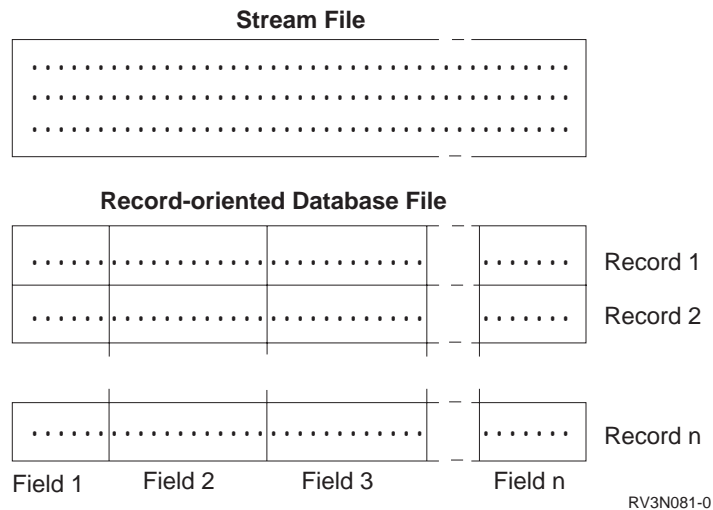


Figure 68. Comparison of a Stream File and a Record-Oriented File

Default C/C++ stream I/O on the iSeries systems is simulated on top of an iSeries database file. Figure 69 illustrates how an iSeries record is mapped to a C/C++ stream. This is simulated stream file processing with iSeries records.

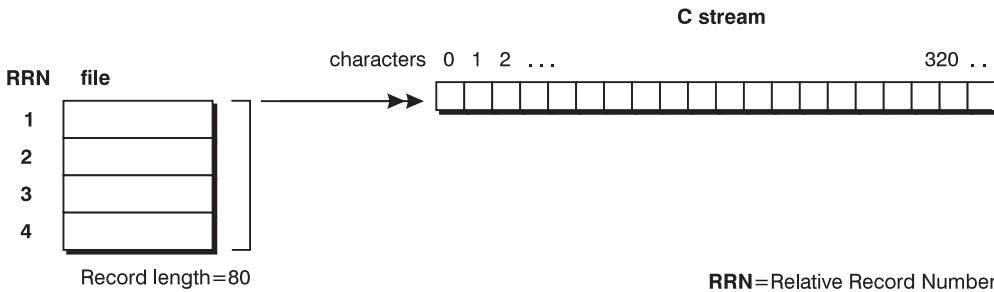


Figure 69. iSeries Records Mapping to a C/C++ Stream File

The differences in structure of stream files and record-oriented files affects how an application is written to interact with them and which type of file is best suited to an application. A **record-arranged file** for example, is well suited for storing customer information, such as name, address, and account balance. These fields can be individually accessed and manipulated using the extensive programming functions of the iSeries system. However, a **stream file** is better suited for storing information such as a customer’s picture, which is composed of a continuous string of bits representing variations in color. Stream files are particularly well suited for storing strings of data such as the text of a document, images, audio, and video.

Text Streams

Text streams contain printable characters and control characters that are organized into lines. Each line consists of zero or more characters and ends with a new-line character (\n). A new-line character is not automatically appended to the end of file.

The ILE C/C++ run time may add, alter, or ignore some special characters during input or output so as to conform to the conventions for representing text in the iSeries environment. Thus, there may not be a one-to-one correspondence between characters written to a file and characters read back from the same file.

Data read from an integrated file system text stream is equal to the data which was written if the data consists only of printable characters and the horizontal tab, new-line, vertical tab, and form-feed control characters.

For most integrated file system stream files, a line consists of zero or more characters, and ends with a carriage-return new-line character combination. However, the integrated file system can have logical links to files on different systems that may use a single line-feed as a line terminator. A good example of this are the files on most UNIX systems.

On input, the default in text mode is to strip all carriage-returns from new-line carriage-return character combination line delimiters. On output, each line-feed character is translated to a carriage-return character that is followed by a line-feed character. The line terminator character sequence can be changed with the CRLN option on `fopen()`.

Note: The *IFSIO option also changes the value for the '\n' escape character value to the 0x25 line feed character. If *NOIFSIO is specified, the '\n' escape character has a value of 0x15.

When a file is opened in text mode, there may be codepage conversions on data that is processed to and from that file. When the data is read from the file, it is converted from the code page of the file to the code page of the application, job, or system receiving the data.

When data is written to an iSeries file, it is converted from the code page of the application, job, or system to the code page of the file. For true stream files, any line-formatting characters (such as carriage return, tab, and end-of-file) are converted from one code page to another.

When reading from QSYS.LIB files end-of-line characters (carriage return and line feed) are appended to the end of the data that is returned in the buffer.

The code page conversion that is done when a text file is processed can be changed by specifying the codepage or CCSID option on `fopen()`. The default is to convert all data read from a file to job's CCSID/code page.

Binary Streams

A **binary stream** is a sequence of characters that has a one-to-one correspondence with the characters stored in the associated iSeries system file. The data is not altered on input or output, so the data that is read from a binary stream is equal to the data that was written. New-line characters have no special significance in a binary stream. The application is responsible for knowing how to handle the data. The `fgets()` function handles new-line characters. Binary files are always created with the CCSID of the job.

Opening Text Stream and Binary Stream Files

Each text stream file and each binary stream file is represented by a file control structure of type `FILE`. This structure is initialized depending on the mode in which the file was opened. Unpredictable results may occur if you attempt to change the file control structure.

The format of `fopen()` is:

```
#include <stdio.h>
FILE *fopen(const char *filename, const char *mode);
```

The *mode* variable is a character string that consists of an open *mode* which may be followed by keyword parameters. The open mode and keyword parameters must be separated by a comma or one or more blank characters.

To open a text stream file, use `fopen()` with one of the following modes:

- | | | |
|-------------------|-------------------|-------------------|
| • <code>r</code> | • <code>w</code> | • <code>a</code> |
| • <code>r+</code> | • <code>w+</code> | • <code>a+</code> |

To open a binary stream file, use `fopen()` with one of the following modes:

- | | | |
|---------------------------|---------------------------|---------------------------|
| • <code>rb</code> | • <code>wb</code> | • <code>ab</code> |
| • <code>rb+ or r+b</code> | • <code>wb+ or w+b</code> | • <code>ab+ or a+b</code> |



To open a binary stream file, use the `open()` member function with `ios::binary`, or any of the following modes:

- `ios::app`
- `ios::ate`
- `ios::in`
- `ios::out`
- `ios::trunc`

Storing Data as a Text Stream or as a Binary Stream

If two streams are opened, one as a binary stream and the other as text stream, and the same information is written to both, the contents of the stream may differ. The following illustrates two streams of different types. The hexadecimal values of the resulting files (which show how the data is actually stored) are not the same.

```

/* Use CRTBND SYSIFCOPT(*IFSIO)                                */
#include <stdio.h>
int main(void)
{
    FILE *fp1, *fp2;
    char lineBin[15], lineTxt[15];
    int x;
    fp1 = fopen("script.bin", "wb");
    fprintf(fp1, "hello world\n");
    fp2 = fopen("script.txt", "w");
    fprintf(fp2, "hello world\n");
    fclose(fp1);
    fclose(fp2);
    fp1 = fopen("script.bin", "rb");
    /* opening the text file as binary to suppress
    the conversion of internal data */
    fp2 = fopen("script.txt", "rb");
    fgets(lineBin, 15, fp1);
    fgets(lineTxt, 15, fp2);
    printf("Hex value of binary file = ");
    for (x=0; lineBin[x]; x++)
        printf("%.2x", (int)(lineBin[x]));
    printf("\nHex value of text file = ");
    for (x=0; lineTxt[x]; x++)
        printf("%.2x", (int)(lineTxt[x]));
    printf("\n");
    fclose(fp1);
    fclose(fp2);

    /* The expected output is:                                */
    /*                                                                */
    /* Hex value of binary file = 888593939640a69699938425      */
    /* Hex value of text file = 888593939640a6969993840d25      */
}

```

Figure 70. Comparison of Text Stream and Binary Stream Contents

As the hexadecimal values of the file contents shows in the binary stream (script.bin), the new-line character is converted to a line-feed hexadecimal value (0x25). While in the text stream (script.txt), the new-line is converted to a carriage-return line-feed hexadecimal value (0x0d25).

Using the Integrated File System

ILE C/C++ primarily supports the iSeries "Root" file system. The "Root" file system is one of the many file systems accessible through the Integrated File System interface. It uses notation similar to that used to access files and directories on UNIX systems, allowing you to access information across multiple platforms in a uniform way.

Care must be taken when transferring files to and from various platforms. Use of a download and upload utility like FTP allows you to specify the correct mapping of characters so your streamed source remains valid on the iSeries platform, even if it has been stored temporarily on other platforms. See "Pitfalls to Avoid" on page 180 for more tips.

Copying Source Files into the Integrated File System

You can copy your main source physical file to an Integrated File System file. Assuming that you used a standard name for your source physical file, use the following command:

```
CPYTOSTMF FROMMBR('/QSYS.LIB/MYLIB.LIB/QCSRC.FILE/QCSRC.MBR') TOSTMF('/home/qcsrc.c')
```

Editing Stream Files

You can edit stream files directly with the Edit File (EDTF) command. There are also three ways that you can edit files to be used with stream files:

- Use Client Access to map the Integrated File System directory as a PC network drive and then use a PC-based editor to edit files in that path as if they were "local" PC files.
- Edit with SEU and then use the Copy to Stream File (CPYTOSTMF) command to move that file from the traditional QSYS file system to a "Root" file system path.
- Place the source in a Source Physical File (SRCPF) with Integrated File System links. (The actual source resides in a QSYS member, but there is a "Root" file system "link" which points to the member.) Use the Add Link (ADDLNK) command to create the link, and thereafter edit the member with SEU, but use the "Root" file system pathname link when you compile.

The SRCSTMF Parameter

The SRCSTMF parameter identifies a source stream file as a path name. Specify the path name of the stream file that contains the ILE C source code that you want to compile. The path name can be either absolutely qualified, or relatively qualified.

For file systems that are case sensitive, the path name may be case sensitive.

An absolutely qualified name starts with '/' or '\'. A / or \ character at the beginning of a path name means that the path begins at the topmost directory, the "root" (/) directory. For example:

```
/Dir1/Dir2/Dir3/UsrFile
```

If the path name does not begin with a / or \ character, the path is assumed to begin at your current directory. For example:

```
MyDir/MyFile
```

is equivalent to

```
/CurrentDir/MyDir/MyFile
```

where MyDir is a subdirectory of your current directory.

There is no support for the tilde (~) character or wildcards (* or ?).

SRCSTMF is mutually exclusive with SRCMBR and SRCFILE. Also, if you specify SRCSTMF, then the compiler ignores TEXT(*SRCMBRTXT). Other values for TEXT are valid.

Header File Search

The compiler uses different search techniques when entering your source file using the source stream file parameters. The compiler no longer uses the library list search method.

Include File Links

ILE C/C++ headers, along with system headers, are located in QSYSINC/H. The links are in the directory /QIBM/INCLUDE.

For example, the links are as follows for assert.h:

- Display Symbolic Link Object link: /QIBM/include/assert.h
- Content of Link: /qsys.lib/qsysinc.lib/h.file/assert.mbr

Include Directive Syntax

Integrated File System (IFS) compiles:

IFS is a hierarchical file system similar to that found on AIX.

When an IFS file specification is used for the root source file (that is, when the SRCSTRMF option is used), all #include directives within that compilation are similarly resolved to the IFS file system. The syntactical variations are:

Table 6. Integrated File System Compiles

#include specification	enclosed in < >	enclosed in " "
filename (e.g., <stdio>)	resolves to [syssearchpath]/filename	resolves to [usrsearchpath]/filename
dir/filename (e.g., <sys/limits.h>)	resolves to [syssearchpath]/dir/filename	resolves to [usrsearchpath]/dir/filename
/dir/filename (e.g., "home/header.h")	resolves to /dir/filename	resolves to /dir/filename

Data Management File System (DM) compiles:

DM is the traditional iSeries monolithic (fixed-depth) file system. It is composed of a number of libraries, which contain objects. There are a fixed set of object types - source files are found within *FILE object types, in "sub-objects" called members. All native iSeries processes have an ordered library list. (aka *LIBL) and in general, iSeries objects are resolved by searching through this library list. The library list has three components, ordered as follows:

- The System Library List (*SYSLIBL): Generally a fixed set of libraries which comprises the operating system.
- The Product Library List (*PRDLIBL): Officially licensed programs typically add themselves to the product library list when run. For example, the C and C++ compilers add their product library QCPPLE to the library list when run.
- The User Library List (*URSLIBL): Libraries that you can configure/order.

When a DM file specification is used for the root source file (that is, when the SRCFILE/SRCMBR options are used), all #include directives within that compilation are similarly resolved to the DM filesystem. The syntactical variations are:

Table 7. Data Management File System Compiles

#include specification	library	file	member
mbr	default search	default file	mbr
mbr.file	default search	file	mbr
file/mbr	default search	file	mbr

Table 7. Data Management File System Compiles (continued)

#include specification	library	file	member
file(mbr)	default search	file	mbr
file/mbr.ext	default search	file	mbr or mbr.ext
lib/file/mbr	lib	file	mbr
lib/file(mbr)	lib	file	mbr

Default library search paths:

Note: When the *SYSINCPATH compile option is specified, then user includes (" ") are treated the same as system includes (< >) when compiled.

- When library is not specified:
 - include specification is enclosed in < >: Search the *LIBL.
 - include specification is enclosed in " ": Check the library containing the root source member; if not found there, then search the *USRLIBL; if still not found, search the *LIBL.
- When library is specified:
 - include specification is enclosed in < >: Search the lib/file/mbr only.
 - include specification is enclosed in " ": Search the user portion of the library list using the file and member names provided; if not found, search for the member in the library/file specified.

Default file:

- Include specification enclosed in < >:
 - C: default file is QCSRC.
 - C++: default file is STD.
- Include specification enclosed in " ":
 - default is the root source file.

mbr.ext

- Include specification enclosed in < >:
 - <file/mbr.h> format resolves to member *mbr* in file *file*. This rule is for Posix support (for example, to be able to include specifications like <sys/limits.h>). The only member names which activate Posix support are extensions of "h" or "H".
 - Otherwise, <file/mbr.ext> resolves to file *file*, and member *mbr.ext*
- Include specification enclosed in " ":
 - "file/mbr.ext" resolves to file *file*, and member *mbr.ext*

Include Search Path Rules

INCDIR (include Directory) Command Parameter:

The Include Directory parameter (INCDIR) works with the Create Module and Create Bound Program compiler commands, allowing you to redefine the path used to locate include header files (with the #include directive) when compiling a source stream file only. The parameter is ignored if the source file's location is not defined as an IFS path via the Source Stream File (SRCSTMF) parameter, or if the full (absolute) path name is specified on the #include directive.

The parameter accepts a list of IFS directories. These directories are inserted to the include search path in the order they are entered.

The include files search path adheres to the following directory search order to locate the file:

Table 8. INCDIR Command Parameter

#include type	Directory Search Order
#include <file_name>	<ol style="list-style-type: none"> 1. If you specify a directory in the INCDIR parameter, the compiler searches for file_name in that directory first. 2. If more than one directory is specified, the compiler searches the directories in the order that they appear on the command line. 3. Searches the directory /QIBM/include.
#include "file_name"	<ol style="list-style-type: none"> 1. Searches the directory where your current source file resides. The current source file is the file that contains the #include "file_name" directive. 2. If you specify a directory in the INCDIR parameter, the compiler searches for file_name in that directory. 3. If more than one directory is specified, the compiler searches the directories in the order that they appear on the command line. 4. Searches the directory /QIBM/include.

For example, if you enter the following value for the INCDIR parameter:

```
Include directory . . . . . INCDIR      '/tmp/dir1'
                  + for more values    './dir2'
```

and with your source stream file you include the following header files:

```
#include "foo.h"
#include <stdio.h>
```

The compiler first searches for a file "foo.h" in the directory where the root source file resides. If the file is found, it is included and the search ends. Otherwise, the compiler searches the directories entered INCDIR, starting with "/tmp/dir1". If the file is found, this file is included. If the directory does not exist, or if the file does not exist within that directory, the compiler continues to search in the subdirectory "dir2" within the current working directory (symbolized by "."). Again, if the file is found, this file is included, otherwise, since the directories in INCDIR path have now been exhausted, the default user include path (/QIBM/include) is used to find the header.

As for <stdio.h>, the same logic is followed in the same order, except the initial search in the root source directory is bypassed.

'INCLUDE' Environment Variable:

The INCLUDE environment variable value contains a path of directories delimited by colons (:).

The INCLUDE and INCLUDEPATH environment variables were supported in the previous release of the compilers, though with different behaviour:

Table 9. INCLUDE Environment Variable

Environment Variable	Release	Compiler	Behavior
INCLUDE	V4R5	ILE C	<ul style="list-style-type: none"> The path overrides the include search path order (i.e., only the INCLUDE path is searched). If you want to search the default include path, it must be explicitly included in the INCLUDE path. If the environment variable is not defined: in a system include search (<>), only the default include path is searched; in a user include search (""), only the current working directory is searched.
INCLUDEPATH	V4R5	ILE C++	<ul style="list-style-type: none"> The path does not override the search order. The INCLUDEPATH directories are searched prior to the default include path. The default path is implicitly included in the INCLUDEPATH path unless the *NOSTDINC option is chosen. If the environment variable is not defined: in a system include search (<>), only the default include path is searched; in a user include search (""), the root source directory and the default include path is searched.
INCLUDE	V5R1	ILE C/C++	<ul style="list-style-type: none"> The environment variable does not override the order. INCLUDE simply has higher priority in the search order than the default include path, but less than INCDIR and the root source's directory (if a user include search).

The resulting include search order including a defined INCLUDE environment variable in V5R1 for both C and C++ compilers is as follows:

Table 10. Include Search Order

#include type	Directory Search Order
#include <file_name>	<ol style="list-style-type: none"> If you specify a directory in the INCDIR parameter, the compiler searches for file_name in that directory first. If more than one directory is specified, the compiler searches the directories in the order that they appear on the command line. If the INCLUDE environment variable is defined, the compiler searches the directories in the order they appear in the INCLUDE path. Searches the directory /QIBM/include.

Table 10. Include Search Order (continued)

#include type	Directory Search Order
#include "file_name"	<ol style="list-style-type: none"> 1. Searches the directory where your current root source file resides. 2. If you specify a directory in the INCDIR parameter, the compiler searches for file_name in that directory. 3. If more than one directory is specified, the compiler searches the directories in the order that they appear on the command line. 4. If the INCLUDE environment variable is defined, the compiler searches the directories in the order they appear in the INCLUDE path. 5. If the INCLUDE environment variable is defined, the compiler searches the directories in the order they appear in the INCLUDE path

Note: This feature is only available for source stream file compiles.

***STDINC/*NOSTDINC Command Options:**

The *STDINC/*NOSTDINC command options work on the CRTCMOD/CRTCPPMOD and CRTBNDC/CRTBNDCPP commands.

The *NOSTDINC option allows you to remove the default include path (/QIBM/include for IFS source stream files; QSYSINC for data management source file members) from the search order, while the *STDINC option retains the default include path at the end of the order. *STDINC is the default.

This option works as did the former SYSINC parameter for data management source file members. The options relate to the old parameter values as follows:

Table 11. Parameter Values

SYSINC values	Equivalent New Command Option
*YES	*STDINC
*NO	*NOSTDINC

***INCDIRFIRST/*NOINCDIRFIRST Command Options:**

The *STDINC/*NOSTDINC command options have been added to the OPTION parameter of the CRTCMOD/CRTCPPMOD and CRTBNDC/CRTBNDCPP commands.

The *INCDIRFIRST option allows you to process the directories listed via the INCDIR parameter first in the search order (that is, before the root source file directory) in a user include search, while the *NOINCDIRFIRST option retains INCDIR directories to their default position in the user include search order as described above.

Note: This option is only valid for source stream file compiles.

For example, if *INCDIRFIRST is selected, the following changes occur to the user include search order:

Table 12. INCDIRFIRST Command Options

#include type	Directory Search Order
#include "file_name"	<ol style="list-style-type: none"> 1. If you specify a directory in the INCDIR parameter, the compiler searches for file_name in that directory. 2. If more than one directory is specified, the compiler searches the directories in the order that they appear on the command line. 3. Searches the directory where your current root source file resides. 4. If the INCLUDE environment variable is defined, the compiler searches the directories in the order they appear in the INCLUDE path. 5. If the *NOSTDINC compiler option is not chosen, search the default include directory /QIBM/include.

***SYSINCPATH/*NOSYSINCPATH Command Options:**

The *SYSINCPATH/*NOSYSINCPATH command options work on the Create Module and Create Bound Program commands.

The *SYSINCPATH option changes the search path of user includes to the system include search path. In affect it is equal to changing the double-qoutes in the user #include directive (#include "file_name") to angle brackets (#include <file_name>). *NOSYSINCPATH turns off this option and is the default value.

Restrictions

Source Stream Files Only:

If you specify SRCSTMF, the *MODULE object contains no source file attribute information.

If the source file is not specified via the Source Stream File parameter, the job's library list (LIBL) is used to define the default include paths. Following the C++ design of INCDIR and the design of the INCLUDE environment variable, this parameter is only used to add "directories" to the default user include path. Since the library list has no concept of the directory file structure, this parameter would be meaningless and is therefore ignored.

Absolute Include Path Name:

If you specify a full (absolute) path name on the #include directive, this option has no effect.

Note:

If you have #include "myinc.h" and a C/C++ source file, you are compiling a source member from the QSYS file system through the SRCSTMF parameter in the compiler command like:

```
CRTCMOD MODULE(MYSOURCE) SRCSTMF('/qsys.lib/goodness.lib/qcppsrc.file/mysource.mbr')
```

ILE C/C++ tries to find something called /qsys.lib/goodness.lib/qcppsrc.file/myinc.h, which is an invalid Integrated File

System filename since .h is not a valid object type in the QSYS file system. If you really want to use a header file that is in the QSYS file system, you must specify a .mbr file extension. For example:

```
#include "/qsys.lib/goodness.lib/h.file/myinc.mbr"
```

or, if you have set the search path appropriately, as shown in “Examples of Using Integrated File System Source” on page 180:

```
#include "myinc.mbr"
```

We recommend that you put source and header files in the Integrated File System to avoid changing your current C++ source code. You may be porting from other platforms which have hierarchical file systems (like the Microsoft® Windows® operating system, Unix, or OS/2), and the Integrated File System is more like those file systems.

Preprocessor Output

If you specify SRCSTMF with OPTION(*PPONLY), then the processor writes a stream file to the current directory with the new extension .i. For example, if you specify SRCSTMF('/home/MOE/mainprogram.c') with OPTION(*PPONLY), then the processor writes output to the current directory as a stream file called mainprogram.i. For this to happen, you need *RW authority to the current directory.

Listing Output

The compiler can send the listing output to a user-specified IFS file, as well as a spool file. The prolog identifies the source file from a path name:

```
Module . . . . . : TEST
Library . . . . . : MOE
Source stream file . . . . . : /home/MOE/src/mainprogram.c
```

It also identifies the include files from their path names. Source stream file is not included in the prolog when SRCFILE() and SRCSTMF() are specified.

```
**** FILE TABLE SECTION ****
0 = hello.cpp
1 = /QIBM/include/iostream.h
2 = /QIBM/include/string.h
```

The listing includes the files specified on any #include directive, and the file specified or implied on the SRCSTMF() or SRCFILE()/SRCMBR() options. This happens for database files and stream file source.

The format of the OUTPUT option is: OUTPUT(*NONE | *PRINT | filename), where *PRINT causes the compiler to send it to a spool file, and filename is the user-specified IFS *file.xxxxx*

Code Pages and CCSIDs

For source physical files, the compiler respects the CCSID of ILE C source. A similar scheme exists for stream file compilation. Stream files have a "code page" attribute.

The compiler converts source files, translating code pages to the root source.

The source stream file may have been entered through a mounted file on an ASCII system. In such a case, the compiler translates from the ASCII codepage that is associated with the stream file (for example, 437) to EBCDIC (for example, 37).

Support for Unicode wide-character literals can be enabled when building your program by specifying `LOCALETYPE(*LOCALEUCS2)` on the compile command. See Chapter 19, "International Locale Support" on page 411 for more information..

You can configure most file transfer utilities to automatically do the conversion to enable ASCII-based file systems to work for producing stream file source.

Pitfalls to Avoid

Any source file created on the workstation with an ASCII editor that deposits an EOF marker at the end of a text file will generate an invalid character warning message when it is compiled with the ILE C/C++ compiler. This includes your main source file. The problem arises when the source file is copied to, or saved in, the "Root" file system on the iSeries . This is because of the translation between ASCII and EBCDIC codepoints.

If you receive an invalid character message referring to the last character of a file, it is likely that you have an EOF marker in the file. One way to avoid this problem is to use an editor which does not add the EOF marker.

Alternatively you can use a File Transfer Protocol (FTP) utility. FTP will result in a "Root" file system file with either codepage 819 or 37. Any of these FTP commands issued to the target iSeries system prior to the put command will result in a file of codepage 819:

- `ascii`
- `quote type a`

If you issue the following command to the target iSeries system prior to the put command, put results in a file with codepage 37 (EBCDIC): `quote type e`. When the file is transferred using FTP to the "Root" file system, the file is created with either codepage 819 or codepage 37 (depending on the previous commands as outlined above) whether the file exists prior to the transfer or not.

Files transferred to an Integrated File System with codepage 37, fail to compile.

Examples of Using Integrated File System Source

The most basic entry of an Integrated File System name does not specify any path information.

Create C++ Module (CRTCPPMOD)

Type choices, press Enter.

Module	> TEST	Name
Library	*CURLIB	Name, *CURLIB
Source file	QCPPSRC	Name
Library	*CURLIB	Name, *CURLIB
Source member	*MODULE	Name, *MODULE
Source stream file	> test.cpp	
Text 'description' *BLANK		

Bottom

F3=Exit F4=Prompt F5=Refresh F10=Additional parameters F12=Cancel
 F13=How to use this display F24=More keys

Without a pathname, the system assumes that your source is located in the current directory. The default current directory is the base "/" directory of the "Root" file system, but your individual user profile may change this default to a different directory. You can change the current directory with the CHGCURDIR command.

Note: The current directory and the current library are separate and distinct entities. Although you can set the current library and the current directory to be the same name, a change in one will not affect the other.

The header files specified in any #include statements in your source will be searched for in the source directory first and then the specified INCDIR directory. For example, if you compile the following source in file /goodness/mysource.cpp:

```
#include "special/mystuff.h"
```

```
class test : public base
{
:
}
```

with the INCDIR value set to /mydir, your included header file is first searched for as /goodness/special/mystuff.h and then /mydir/special/mystuff.h.

Using Stream I/O

The following sections describe stream I/O changes for Version 5 Release 1.

Large Files

Within the C or C++ run-time, stream I/O for files up to two gigabytes in size is enabled by specifying the *IFSIO option on the system interface keyword (SYSIFCOPT) on the Create Module or Create Bound Program command prompt. The format of the SYSIFCOPT command is:

```
CRTCPPMOD MODULE(QTEMP/IFSIO) SRCFILE(QCLE/QACSRC) SYSIFCOPT(*IFSIO)
CRTBNDCPP PGM(QTEMP/IFSIO) SRCFILE(QCLE/QACSRC) SYSIFCOPT(*IFSIO)
```

When this option is specified, the compiler defines the __IFS_IO__ macro. When __IFS_IO__ is defined, prototypes associated with stream I/O in <stdio.h> are no

longer defined. The header file <ifs.h> is included by <stdio.h>, which declares all structure and prototypes associated with the integrated file system enabled C stream I/O.

The 64-bit version of the Integrated File System interface lets you use ILE C stream I/O with files up to and greater than two gigabytes in size. (C++ stream I/O for files greater than two gigabytes is not supported.) To enable the 64-bit interface, specify the *IFS64IO option with the SYSIFCOPT keyword on the CRTCPMOD or CRTBNDCPP command prompt. When this option is specified, the compiler defines the __IFS64_IO__ macro which, for example, remaps the open() function to an open64() function to allow 64-bit indexing..


Open Mode



The fstream(), ifstream(), and ofstream() classes have a new open mode ios::text, which opens the file in text mode.

By default, I/O streams are opened in binary mode (not in text mode, as stated in the Version 3 Release 7 books).

Line-End Characters

- If the input or output is unformatted (for example, via the read() or write() method), newline (\n) characters are *not* expanded to \r\n on output and \r characters are *not* stripped out on input.
-  If the input or output is formatted (via the >> or << operator), newline (\n) characters are *not* expanded to \r\n on output but any \r characters *are* stripped out on input

If you want carriage return (\r) characters to be added, use the fopen() function with cr\n=Y (the default).

Part 5. Working with iSeries File Systems and and Devices

This part describes:

- Retrieve external file descriptions
- Work in disconnected mode
- Include externally described physical and logical database files
- Use physical and logical database files and distributed data
- Use commitment control
- Use display files, printer files, ICF files, tape files, diskette files, and save files
- Use the device attributes feedback area

Chapter 10. Using Externally Described Files in Your Programs

This chapter describes how to use:

- Externally described physical and logical database files
- Externally described device files
- Multiple record formats
- Zoned decimal and packed decimal data

Externally described files are files that have their field descriptions stored as part of the file. The description includes information about the type of file (such as data or device), record formats, and a description of each field and its attributes.

You can create an externally described database file using the SQL/400 database, the Interactive Data Definition Utility (IDDU) using DDS for externally described files, or with Data Description Specifications (DDS). The ILE C preprocessor automatically creates C structure typedefs from external file descriptions when you use the `#pragma mapinc` directive with the `#include` directive.

The `#pragma mapinc` directive only identifies the file formats and fields to the compiler; it does not include the file description in the ILE C program. To include a file description, the `#include` directive must be coded in the ILE C program.

You refer to the include-name parameter of the `#pragma mapinc` directive on the `#include`. The `#include` directive must be coded after the `#pragma mapinc` directive in your source program.

For example, to include a type definition of the input fields for the record format FMT from the file EXAMPLE/TEST, the following statements must appear in your program in the order shown below:

```
#pragma mapinc("tempname","EXAMPLE/TEST(FMT)","input","d",",")
#include "tempname"
```

Typedefs

In C programs, for each format specified on the `#pragma mapinc` directive, the compiler creates a C typedef of type struct describing the fields in the external file. You indicate the type of structure definitions to be included in your ILE C/C++ program by the element you select on the *options* parameter. A header description is also created which contains information about the external file.

Note: `#pragma mapinc` will be phased out in future releases. It is recommended that you use the GENCSRC utility.

You can compile your ILE C/C++ program with `OUTPUT(*PRINT)` `OPTION(*SHOWUSR)` to see the typedefs in your compiler listing. The type definitions are also generated in the listing if you specify `OUTPUT(*PRINT)` `OPTION(*SHOWINC)`.

Header Description

The header description for each format contains the following information:

- File and library name of the external file
- File type (physical, logical, device)
- Date the file was created
- Record format name
- Record format level ID (level checking information).

For example, the following directives are used to create the header shown below:

```
#pragma mapinc("payroll","example/test(fmt1)","input","")
#include "payroll"

/* ----- */
/* PHYSICAL FILE: EXAMPLE/TEST */
/* FILE CREATION DATE: 93/09/01 */
/* RECORD FORMAT: FMT1 */
/* FORMAT LEVEL IDENTIFIER: 371E00A681EA7 */
/* ----- */
```

Figure 71. Header Description

The following is an example of a type definition of type structure:

```
typedef struct {
    .
    .
    .
} LIBRARY_FILE_FORMAT_tag_t;
```

Parameters of the #pragma mapinc directive are used to create the name of the created type. LIBRARY, FILE, and FORMAT are the library-name, file-name, and format-name specified on the #pragma mapinc directive. These names are folded to uppercase unless quoted names are used. The library and file names can be replaced with your own prefix-name as specified on the #pragma mapinc directive.

Any characters that are not recognized as valid by the C language that appear in library and file names are translated to the underscore (_) character.

Note: Do not use the special characters #, @, or \$ in library and file names. If these characters are used in library and file names, they are also translated to the underscore (_) character.

The tag on the structure name indicates the type of fields that are included in the structure definition. The possible values for tag are:

Field Type	Tag	Field Type	Tag
input	i	key	key
output	o	indicators	indic
both	both	nullflds	nmap/nkmap

Unlike the naming convention used for other listed field types, if field type lvlchk is specified, the name of the array of structure type created is _LVLCHK_T.

To include external file descriptions for more than one format, specify more than one format name (format1 format2) or (*ALL) on the #pragma mapinc directive. A header description and typedefs are created for each format.

When the lname option is specified and the filename in the #pragma mapinc directive is greater than 10 characters in lengths, a system generated 10-character name will be used in the typedefs generated by the compiler.

Level Checking

If you specify the lvlchk option on the #pragma mapinc directive, in addition to generating the type _LVLCHK_T (array of structures), a variable of type _LVLCHK_T is also generated. It is also initialized so that each array element contains the level check information for the corresponding formats specified on the #pragma mapinc. The last two array elements are always empty strings, one for each field of the structure.

The name of the variable is LIBRARY_FILE_INCLUDE_lvlchk, where LIBRARY, FILE, and INCLUDE are the library_name, file_name and include_name, respectively. The level check information can be used to perform a level check on the file when it is opened. If you specify the lvlchk keyword on the _Ropen varparm parameter and the make-up of the file is changed, the file pointer on the _Ropen returns NULL and CPF4131 is issued.

Example

The following example shows the #pragma mapinc directive and the lvlchk option to perform a level check on a file when it is opened.

```
/* This program illustrates how to use level check information.      */
/* This example uses ILE C record I/O. See the ILE C                */
/* Programmer's Reference for descriptions of the record I/O        */
/* functions.                                                        */
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>
#pragma mapinc("DD3FILE","MYLIB/T1520DD3(purchase)","key lvlchk","_P")
#include "DD3FILE"
int main(void)
{
    _RFILE    *in;
    char      new_purchase[21] = "PEAR      1002022244";
    /* Open the file for processing in keyed sequence. File is created */
    /* with the default access path.                                    */
```

Figure 72. ILE C Source Using the lvlchk Option (Part 1 of 2)

```

        if ( (in = _Ropen("MYLIB/T1520DD3", "rr+ varparm = (lvlchk)",
            &MYLIB_T1520DD3_DD3FILE_lvlchk)) == NULL)
        {
            printf("Open failed\n");
            exit(1);
        }

/* Update the first record in the keyed sequence. The function */
/* _Rlocate locks the record. */
    _Rlocate(in, NULL, 0, __FIRST);
    _Rupdate(in, new_purchase, 20);
/* Force the end of data. */
    _Rfeod(in);
    _Rclose(in);
}

```

Figure 72. ILE C Source Using the *lvlchk* Option (Part 2 of 2)

The following example contains the DDS in the file T1520DD3 in the library MYLIB.

```

A          R PURCHASE
A          ITEMNAME      10
A          SERIALNUM     10
A          K SERIALNUM

```

Figure 73. T1520DD3 — DDS Source for Program

The DDS part of the program listing is as follows:

```

/* -----*/
/* PHYSICAL FILE: MYLIB/T1520DD3 */
/* FILE CREATION DATE: 93/09/02 */
/* RECORD FORMAT: PURCHASE */
/* FORMAT LEVEL IDENTIFIER: 322C4B361172D */
/* -----*/
typedef _Packed struct {
    char SERIALNUM[10];
                                /* DDS - ASCENDING*/
                                /* STRING KEY FIELD*/
}MYLIB_T1520DD3_PURCHASE_key_t;
typedef _Packed struct {
    unsigned char format_name[10];
    unsigned char sequence_no[13];
} _LVLCHK_T[];
_LVLCHK_T MYLIB_T1520DD3_DD3FILE_lvlchk = {
    "PURCHASE ", "322C4B361172D",
    "", "" };

```

Figure 74. Ouput Listing from the Program

Record Format Name

A **record format** describes all the fields and the arrangement of these fields within a record. You can include a record format from an externally described file in your Integrated Language Environment program by providing its name on the `#pragma mapinc` directive. You can provide more than one format name, or you can specify the special value `*ALL` to include all record formats from the file.

If the file you are working with contains more than one record format, set the format for subsequent I/O operations with the `_Rformat()` function.

Record format functions are useful when working with display, ICF, and printer files. Logical files can also contain more than one record format.

The record format name for a device file defaults to blank unless you explicitly set it to a name with `_Rformat()`. You can reset the format name to blank by passing a blank name to `_Rformat()`.

If the record format does not contain fields that match the option specified (input, output, both, key, indicators or nullflds) on the `#pragma mapinc` directive, the following comment appears after the header description:

```
/* FORMAT HAS NO FIELDS OF REQUIRED TYPE                                */
```

Note: Do not use #, @, or \$ in record format names. These characters are not allowed in Integrated Language Environment identifiers and cannot appear in a typedef name. If you have record format names that contain #, @ or \$, these characters are translated to the lowercase characters p, a, and d, respectively.

Record Field Names

All DDS keywords are supported by the Integrated Language Environment library and compiler. The actual comment that is associated with the TEXT keyword is translated to uppercase in the typedef that is generated. The ALIAS keyword is supported and brings the alias field name into the typedef that is generated.

Some of the special characters that are supported in DDS variable names are not supported by the Integrated Language Environment compiler and library. If you use the special characters @, #, or \$ in a field name, those characters are changed to lowercase a, p and d in the typedef that is generated. If the format names contain C characters that are not valid, they are translated to the underscore (_) character.

Alignment of Fields in C Structures

All fields defined in ILE C/C++ structures are aligned on their natural boundaries. For example, int fields are 4 bytes long and are stored on 4-byte boundaries. If you create a file that is externally described, the system does not enforce boundary alignment of the externally described data. The structure may need to be packed because packed structures match the alignment of the externally described data.

If the fields defined in the DDS are aligned (for example, all are character fields), you can use the typedef that is generated without packing the structure.

To avoid an alignment problem, specify the `_P` option to generate a packed structure. For example, to include a packed type definition structure of input and key fields for the record format `custrec` from the file `EXAMPLE/CUSTOMSTL`, the following statements must appear in your program in the order shown below:

```
#pragma mapinc("custmf","EXAMPLE/CUSTOMSTL(custrec)","input key","_P")
...
#include "custmf"
```

Using Externally Described Physical and Logical Database Files

To include external database file descriptions, use the input, both, key, nullflds, or lvlchk option on the #pragma mapinc directive. If you specify either the output or indicators option, an error message is issued.

You can include external file descriptions for Distributed Data Management (DDM) files using the same method described for database files if you specify either the input, key, or both option. If you specify output or indicators, an error message is issued.

Input and Both Fields

Input and output buffers for database files have the same format. When you specify input, the fields that are defined as either INPUT or BOTH in the externally described database file are included in the typedef. When you specify both, the fields that are defined as either INPUT, OUTPUT, or BOTH are included in the typedef.

If all the fields are defined as BOTH or INPUT, only one typedef structure is generated in the typedef.

Key Fields

To include a separate structure type definition for the key fields in a format, specify the key option on the #pragma mapinc directive. Comments are listed beside the fields in the structure definition to indicate how the key fields are defined in the externally described file.

Example

The following ILE C program contains the #pragma mapinc directive to include the externally described database file CUSMSTL:

```
#pragma mapinc("custmf","example/cusmstl(cusrec)","both key","d")
#include "custmf"
```

The following example contains the DDS for the file T1520DD8 in the library MYLIB.

```
A* CUSTOMER MASTER FILE -- T1520DD8
  A          R CUSREC                TEXT('Customer master record')
  A          CUST          5         TEXT('Customer number')
  A          NAME         20         TEXT('Customer name')
  A          ADDR         20         TEXT('Customer address')
  A          CITY         20         TEXT('Customer city')
  A          STATE         2         TEXT('State abbreviation')
  A          ZIP           5  0       TEXT('Zip code')
  A          ARBAL        10  2       TEXT('Accounts receivable balance')
  A          K CUST
A*
A*
```

Figure 75. T1520DD8 — DDS Source for Customer Records

Program T1520EDF uses the `#pragma mapinc` directive to generate the file field structure that is defined in T1520DD8.

```
/* This program contains the #pragma mapinc directive to          */
/* include the externally described database file T1520DD8.       */
/* This program reads customer information from a terminal and issues */
/* a warning message if the customer's balance is less than $1000. */

#include <stdio.h>
#include <recio.h>
#include <stdlib.h>
#include <string.h>
#include <decimal.h>

#pragma mapinc("custmf","QGPL/T1520DD8(cusrec)","both key","_P")
#include "custmf"

int main(void)
{

    /* Declare x of data structure type QGPL_T1520DD*_CUSREC_both_t. */
    /* The data structure type was defined from the DDS specified.    */

    QGPL_T1520DD8_CUSREC_both_t x;

    /* Get information from entry.                                     */

    printf("Please type in the customer name (max 20 char).\n");
    gets(x.NAME);
    printf("Please type in the customer balance.\n");
    scanf("%D(10,2)", &x.ARBAL);

    /* Prints out warning message if x.ARBAL<1000.                  */

    if (x.ARBAL<1000)
    {
        printf("%s has a balance less than $1000!\n", x.NAME);
    }
}
```

Figure 76. T1520EDF — ILE C Source to Include an Externally Described Database File

The typedefs are created in your ILE C source listing that is based on the `#pragma` directive that is specified in the ILE C source program.

The output is as follows:

```
Please type in the customer name (max 20 char).
> James Smith
Please type in the customer balance.
> 250.58
James Smith has a balance less than $1000!
Press ENTER to end terminal session.
```

The DDS part of the program listing is as follows:

```

/* -----*/
/* PHYSICAL FILE: MYLIB/T1520DD8 */
/* FILE CREATION DATE: 93/08/14 */
/* RECORD FORMAT: CUSREC */
/* FORMAT LEVEL IDENTIFIER: 4E9D9ACA60E00 */
/* -----*/
typedef _Packed struct {
    char CUST[5]; /* Customer number */
    char NAME[20]; /* Customer name */
    char ADDR[20]; /* Customer address */
    char CITY[20]; /* Customer city */
    char STATE[2]; /* State abbreviation*/
    decimal(5,0) ZIP; /* Zip code */
    decimal(10,2) ARBAL; /* Accounts receivable balance*/
}MYLIB_T1520DD8_CUSREC_both_t;
typedef _Packed struct {
    char CUST[5];
}MYLIB_T1520DD8_CUSREC_key_t;
/* DDS - ASCENDING*/
/* STRING KEY FIELD*/

```

Figure 77. Ouput Listing from Program T1520EDF — Customer Master Record

Using Externally Described Device Files

To include external device file descriptions, use the input, output, both, and/or indicators *options* on the #pragma mapinc directive. Device files do not contain key fields. Therefore, you cannot specify the key option with device files.

Input Fields

You specify the input option when you want to include the fields that are defined as INPUT or BOTH in the externally described device file. Response indicators are included when the DDS keyword INDARA is not specified.

When this DDS is included in your ILE C/C++ program:

```

#pragma mapinc("test","example/phonelist(phone)","input","")
#include "test"
A          R PHONE
A          CF03(03 'EXIT')
A          1 35'PHONE BOOK'
A          DSPATR(HI)
A          7 28'Name:'
A          NAME          11A I 7 34
A          ADDRESS      20A I 9 34
A          PHONE_NUM     8A I 11 34
A          23 34'F3 - EXIT'
A          DSPATR(HI)

```

Figure 78. DDS Source for a Display File

The following struct is generated:

```
/* -----*/
/* DEVICE FILE: EXAMPLE/PHONELIST */
/* FILE CREATION DATE: 93/09/01 */
/* RECORD FORMAT: PHONE */
/* FORMAT LEVEL IDENTIFIER: 10D2D0DB2BEE8 */
/* -----*/
typedef struct {
    char IN03; /* EXIT */
    char NAME[11];
    char ADDRESS[20];
    char PHONE_NUM[8];
}EXAMPLE_PHONELIST_PHONE_i_t;
```

Figure 79. Structure Definition for a Display File

Output Fields

Specify the output option when you want to include fields that are defined as OUTPUT or BOTH in the externally described device file. Option indicators are included when the DDS keyword INDARA is not specified.

Both Fields

When you specify both, two typedef structs are generated. One typedef contains all fields defined as INPUT or BOTH; the other contains all fields defined as OUTPUT, or BOTH. One typedef structure is generated for each format that is specified when all fields are defined as BOTH and a separate indicator area is not specified. Option and response indicators are included in the typedef structs if the keyword INDARA is not specified in the external file description.

If you are including the external file description for only one record format, a typedef union is automatically created containing the two typedefs. The name of this typedef union is LIBRARY_FILE_FMT_both_t. If you specify a union-type-name on the #pragma mapinc directive, the name that is generated is union-type-name_t.

A				INDARA
A	R	FMT		
A				CF01(50)
A				CF02(51)
A				CF03(99 'EXIT')
A			1	35'PHONE BOOK'
A				DSPATR(HI)
A			7	28'Name: '
A	NAME	11A	I	7 34
A			9	25'Address: '
A	ADDRESS	20A	0	9 34
A			11	25'Phone #: '
A	PHONE_NUM	8A	0	11 34
A			23	34'F3 - EXIT'

Figure 80. DDS Source for a Device File

When the DDS shown above is included in your ILE C program, the following struct is generated:

```
#pragma mapinc("example/screen1","example/test(fmt)","both","d")
#include "example/screen1"

/* -----
/* DEVICE FILE: EXAMPLE/TEST
/* FILE CREATION DATE: 93/09/01
/* RECORD FORMAT: FMT
/* FORMAT LEVEL IDENTIFIER: 371E00A681EA7
/* -----
/* INDICATORS FOR FORMAT FMT
/* INDICATOR 50
/* INDICATOR 51
/* INDICATOR 99
/* -----
typedef struct {
    char NAME[11];
}EXAMPLE_TEST_FMT_i_t;
typedef struct {
    char ADDRESS[20];
    char PHONE_NUM[8];
}EXAMPLE_TEST_FMT_o_t;
typedef union {
    EXAMPLE_TEST_FMT_i_t    EXAMPLE_TEST_FMT_i;
    EXAMPLE_TEST_FMT_o_t    EXAMPLE_TEST_FMT_o;
}EXAMPLE_TEST_FMT_both_t;
```

Figure 81. Structure Definitions for a Device File

This shows the structure definitions that are created when the format FMT in the device file EXAMPLE/TEST is included in your program. The external file description contains three indicators IN50, IN51, and IN99, and the keyword INDARA. The indicators will appear as comments and will not be included in the structure as the option "indicators" was not specified in the #pragma mapinc.

Indicator Field

Indicators for a record format are allowed only for device files, and can be defined as a separate indicator structure or as a member in the record format structure.

Separate Indicator Area

To use indicators as a separate structure you must specify:

- the INDARA keyword in the external description of the file
- the "indicators" option on the #pragma mapinc directive.
- use "indicators=y" when opening the file.

You must also set the address of the separate indicator area by using the `_Rindara()` function for record files, before performing I/O operations. If you specify indicators on the #pragma mapinc directive and do not use the keyword INDARA in your external file description, you will receive a warning message at compile time.

If indicators are requested and exist in the format, a 99-byte structure definition is created that contains a field declaration for each indicator defined in the DDS. The

name of each field is INXX, where XX is the DDS indicator number. The sequence of bytes between indicators is defined as INXX_INYY, where XX is the first undefined byte and YY is the last undefined byte in a sequence.

```

A                                INDARA
A      R FMT
A                                CF01(50)
A                                CF02(51)
A                                CF03(99 'EXIT')
A                                1 35'PHONE BOOK'
A                                DSPATR(HI)
A                                7 28'Name:'
A      NAME      11A I 7 34
A                                9 25'Address:'
A      ADDRESS   20A 0 9 34
A                                11 25'Phone #:'
A      PHONE_NUM 8A 0 11 34
A                                23 34'F3 - EXIT'
```

Figure 82. DDS Source for Indicators

```
#pragma mapinc("example/temp","exindic/test(fmt)","indicators","")
#include "example/temp"
```

When this DDS is included in your ILE C/C++ program, the following struct is generated:

```

/* ----- */
/* DEVICE FILE: EXINDIC/TEST */
/* FILE CREATION DATE: 93/09/01 */
/* RECORD FORMAT: FMT */
/* FORMAT LEVEL IDENTIFIER: 371E00A681EA7 */
/* ----- */
/* INDICATORS FOR FORMAT FMT */
/* INDICATOR 50 */
/* INDICATOR 51 */
/* INDICATOR 99 */
/* ----- */
typedef struct {
    char IN01_in49[49];          /* UNUSED INDICATOR(S) */
    char IN50;
    char IN51;
    char IN52_in98[47];         /* UNUSED INDICATOR(S) */
    char IN99;
}EXINDIC_TEST_FMT_indic_t;
```

Figure 83. Structure Definition for Indicators

This shows a typedef of a structure for the indicators in the format FMT of the file EXINDIC/TEST. The external file description contains three indicators: IN50, IN51, and IN99. The keyword INDARA is also specified in the DDS for the file.

If indicators are defined for a record format and the indicators option is not specified on the #pragma mapinc directive, a list of the indicators in the DDS is included as a comment in the header description. The following shows the header

description created when the file description for the file EXINDIC/TEST is included in your program and the indicators option is not specified on the #pragma mapinc directive.

```
/* ----- */
/* DEVICE FILE: EXINDIC/TEST */
/* CREATION DATE: 93/09/01 */
/* RECORD FORMAT: FMT */
/* FORMAT LEVEL IDENTIFIER: 371E00A681EA7 */
/* ----- */
/* INDICATORS FOR RECORD FORMAT FMT */
/* INDICATOR 50 */
/* INDICATOR 51 */
/* INDICATOR 99 */
/* ----- */
```

Figure 84. Header Description Showing Comments for Indicators

Indicator in the File Buffer

If you do not specify the INDARA keyword in DDS, the indicators are part of the file buffer being read or written. Only indicators that are used are declared in the type definition of the structure. They are declared as char, and created when one of the input, output, or both *options* is specified on the #pragma mapinc directive.

Using Multiple Record Formats

To include multiple formats in a logical file, specify more than one record format name or (*ALL) on the #pragma mapinc directive. If you specify multiple formats, a header description and typedef is created for each format. If you specify a union-type-name, a union typedef is created.

The typedef union contains structure definitions created for each format. Structure definitions that are created for key fields when the key option is specified are not included in the union definition. The name of the union definition is union-type-name_t. The name you provide for the union-type-name is not folded to uppercase.

The following shows the typedefs created for a logical file with two record formats with the BOTH and KEY *options* specified. A typedef union with the tag buffer_t is also generated.

```
#pragma mapinc("pay", "lib1/pay(fmt1 fmt2)", "both key", "", "buffer", "Pay")
#include "pay"
```



```

/* -----*/
/* LOGICAL FILE: PAY */
/* FILE CREATION DATE: 93/09/01 */
/* RECORD FORMAT: FMT1 */
/* FORMAT LEVEL IDENTIFIER: 371E00A681EA7 */
/* -----*/
typedef struct {
    .
    .
    .
}Pay_FMT1_both_t;

typedef struct {
    .
    .
    .
}Pay_FMT1_key_t;

/* -----*/
/* LOGICAL FILE: PAY */
/* FILE CREATION DATE: 93/09/01 */
/* RECORD FORMAT: FMT2 */
/* FORMAT LEVEL IDENTIFIER: 371E00A681EA7 */
/* -----*/
typedef struct {
    .
    .
    .
}Pay_FMT2_both_t;

typedef struct {
    .
    .
    .
}Pay_FMT2_key_t;

typedef union {
    Pay_FMT1_both_t;          Pay_FMT1_both;
    Pay_FMT2_both_t;          Pay_FMT2_both;
}buffer_t;

```

Figure 85. Structure Definition for Multiple Formats

Note: A typedef union is not created for the key fields.

If you specify *ALL, or more than one record format on the format-name parameter, structure definitions for multiple formats are created.

If you specify multiple formats, and the input, or output option, one structure is created for each format. The following shows the structure definitions that are created when you include the following statements in your program. The device file TESTLIB/FILE contains two record formats, FMT1, and FMT2. Each record format has fields defined as OUTPUT in its file description.

```

#pragma mapinc("example","testlib/file(fmt1 fmt2)","output","z","unionex")
#include "example"

```

```

/* ----- */
/* DEVICE FILE: TESTLIB/FILE */
/* FILE CREATION DATE: 93/09/01 */
/* RECORD FORMAT: FMT1 */
/* FORMAT LEVEL IDENTIFIER: 371E00A681EA7 */
/* ----- */
typedef struct {
    .
    .
    .
}TESTLIB_FILE_FMT1_o_t;

/* ----- */
/* DEVICE FILE: TESTLIB/FILE */
/* CREATION DATE: 93/09/01 */
/* RECORD FORMAT: FMT2 */
/* FORMAT LEVEL IDENTIFIER: 371E00A681EA8 */
/* ----- */
typedef struct {
    .
    .
    .
}TESTLIB_FILE_FMT2_o_t;

typedef union {
    TESTLIB_FILE_FMT1_o_t    TESTLIB_FILE_FMT1_o;
    TESTLIB_FILE_FMT2_o_t    TESTLIB_FILE_FMT2_o;
}unionex_t;

```

Figure 86. Structure Definitions for a Device File

When both are specified as an option, two structure definitions are created for each format. The following shows the structure definitions created when you include two formats, FMT1 and FMT2, for the device file EXAMPLE/TEST and specify the both option:

```

#pragma mapinc("test","example/test(fmt1 fmt2)","both","z","unionex")
#include "test"

```

If all the fields are defined as BOTH and there are to be no indicators in the typedef struct, only one typedef struct is generated for each format specified. The following shows a separate typedef structure for input and output fields.

```

/* ----- */
/* DEVICE FILE: EXAMPLE/TEST */
/* CREATION DATE: 93/09/01 */
/* RECORD FORMAT: FMT1 */
/* FORMAT LEVEL IDENTIFIER: 371E00A681EA7 */
/* ----- */
typedef struct {
    .
    .
    .
}EXAMPLE_TEST_FMT1_i_t;
typedef struct {
    .
    .
    .
}EXAMPLE_TEST_FMT1_o_t;
/* ----- */
/* DEVICE FILE: EXAMPLE/TEST */
/* CREATION DATE: 93/09/01 */
/* RECORD FORMAT: FMT2 */
/* FORMAT LEVEL IDENTIFIER: 371E00A681EA8 */
/* ----- */
typedef struct {
    .
    .
    .
}EXAMPLE_TEST_FMT2_i_t;
typedef struct {
    .
    .
    .
}EXAMPLE_TEST_FMT2_o_t;
typedef union{
    EXAMPLE_TEST_FMT1_i_t      EXAMPLE_TEST_FMT1_i;
    EXAMPLE_TEST_FMT1_o_t      EXAMPLE_TEST_FMT1_o;
    EXAMPLE_TEST_FMT2_i_t      EXAMPLE_TEST_FMT2_i;
    EXAMPLE_TEST_FMT2_o_t      EXAMPLE_TEST_FMT2_o;
}unionex_t;

```

Figure 87. Structure Definitions for BOTH Option

Using Packed Decimal and Zoned Decimal Data

The ILE C/C++ compiler and library supports packed decimal data. The `d` option of the `#pragma mapinc` directive causes the compiler to generate packed decimal variables for any packed decimal fields defined in the DDS file. If you use the `p` option of the `#pragma mapinc` directive, character arrays are generated to store the packed decimal values.

These character arrays then need to be converted to an ILE C/C++ numeric data type to be used in the ILE C/C++ program. If neither the `d` or `p` option is specified, the `d` option is the default. See Chapter 15, “Using Packed Decimal Data in Your C Programs” on page 309 for examples in using packed decimal data types.

Note: You must include the `<decimal.h>` header file if you have a DDS file with packed decimal fields defined. You must also use the `#pragma mapinc` directive `d` option in your ILE C/C++ source code.

The ILE C/C++ compiler and library do not support zoned decimal data. If there are zoned fields, `#pragma mapinc` (and `GENCSRC`) by default, convert them into `*CHAR` type (which makes this parameter senseless).

To convert packed decimal data that is stored in character arrays to integer or floating point (double) and vice versa, the functions `QXXDTP()`, `QXXITP()`, `QXXPTD()`, `QXXPTOI()` can be used.

To convert zoned decimal data that is stored in character arrays to integer or floating point (double) and vice versa, the functions `QXXDTZ()`, `QXXITZ()`, `QXXZTD()`, `QXXZTOI()` can be used.

The `MI cpyrv()` function can also be used to convert packed or zoned decimal data to an ILE C/C++ numeric data type. It can be used to convert an ILE C/C++ numeric data type to packed or zoned decimal data.

The conversion functions are included with the ILE C/C++ compiler so that EPM C code that uses these functions can be maintained.

If you are doing database I/O operations, you can use a logical file with integer or floating point fields to redefine packed and zoned fields in your physical file. When you perform an input, or output operation through the logical file, the iSeries 400 system converts the data for you automatically.

Using Long Names for Files

The `#pragma mapinc` directive supports file names up to 128 characters long and record field names up to 30 characters long. The `lname` option was added to `#pragma mapinc` to support SQL long name formats. SQL long names map to a 10 character short file name, which consist of the first 5 characters of the name, followed by a 5 digit unique number.

For example, the system short name for `LONGSQLTABLENAME` is `LONGS00001`.

Long record field names are not mapped to a 10 character short name. When `lname` option is specified it is assumed that the long name format for the file name is being used. If the file name has more than 10 characters then internally this name is converted to the associated short name. This short name is used to extract the external file definition. When a regular short name of 10 characters or less is specified, no conversion occurs.

The `#pragma mapinc` directive uses the 30 character record field names in the typedefs that are generated, with or without the `lname` option that is specified. For the filenames that are specified using a long name format, the typedefs generated use the associated regular 10 character short filename.

Chapter 11. Using Database Files and Distributed Data Management Files In Your Programs

This chapter describes how to:

- Copy data from one file to another using an arrival sequence access path
- Update data in a record file by using a keyed sequence access path
- Read and print records from a data file
- Specify commitment control conditions

Database Files and Distributed Data Management Files

A **database file** contains data that is stored permanently on the system. The object type is *FILE.

Database files can be created and used as either physical files or logical files. Database files can contain either data or source statements.

ILE C/C++ programs access files on remote systems through **distributed data management** (DDM). DDM allows application programs on one system to use files that are stored on a remote system as database files. No special statements are required in ILE C/C++ programs to support DDM files.

A DDM file is created by a user or program on a local (source) system. This file (with object type *FILE) identifies a file that is kept on a remote (target) system. The DDM file provides the information that is needed for a local iSeries 400 to locate a remote iSeries 400 and to access the data in the target file.

Physical Files and Logical Files

Physical files contain the actual data that is stored on an iSeries system, and a description of how data is to be presented to or received from a program. They contain only one record format, and one or more members.

Records in database files can be described using either a field level description or record level description.

A **field-level description** describes the fields in the record to the system. Database files that are created with field level descriptions are referred to as **externally described files**.

A **record-level description** describes only the length of the record, and not the contents of the record. Database files that are created with record level descriptions are referred to as **program-described files**. This means that your ILE C/C++ program must describe the fields in the record.

An ILE C/C++ program can use either externally described or program-described files. If it uses an externally described file, the ILE C/C++ compiler can extract information from the externally described file, and automatically include field information in your program. Your program does not need to define the field information. For further information see Chapter 10, "Using Externally Described Files in Your Programs" on page 185.

A physical file can have a keyed sequence access path. This means that data is presented to an ILE C/C++ program in a sequence that is based on one or more key fields in the file.

Logical files do not contain data. They contain a description of records that are found in one or more physical files. A logical file is a view or representation of one or more physical files. Logical files that contain more than one format are referred to as **multi-format** logical files.

If your program processes a logical file which contains more than one record format, you can use the `_Rformat()` function to set the format you wish to use.

Some operations cannot be performed on logical files. If you open a logical file for stream file processing with open modes `w`, `w+`, `wb` or `wb+`, the file is opened but not cleared. If you open a logical file for record file processing with open modes `wr` or `wr+`, the file is opened but not cleared.

Records in iSeries database files can be described using either a field level description or record level description.

The field-level description of the record includes a description of all fields and their arrangement in this record. Since the description of the fields and their arrangement is kept within a database file and not in your ILE C/C++ program, database files created with a field-level description are referred to as **externally described files**. See Chapter 10, "Using Externally Described Files in Your Programs" on page 185.

To define externally described database files, use one of the following:

- DB2® database
- Interactive Data Definition Utility (IDDU)
- Data Descriptive Specifications (DDS) source

A **data description specification** is a description of a database file that is entered into the system in a fixed form, and is used to create files. This description is composed of one or more record formats that define the fields that make up the record. It can also include access path information that determines the order in which records are retrieved from the file.

Data Files and Source Files

A **data file** contains actual data.

Records in data files are grouped into members. All the records in a file can be in one member, or they can be grouped into different members. Most database commands and operations by default assume that database files which contain data have only one member. This means that when your ILE C program works with database files containing data you do not need to specify the member name for the file unless your file contains more than one member.

Usually, database files that contain source programs are made up of more than one member. Organizing source programs into members within database files allows you to better manage your programs. These **source members** contain source statements that the iSeries system uses to create iSeries objects. For example, a source member which contains C++ statements is used to create a program object.

Access Paths

Access paths describe the logical order of records in a file. There are two types of access paths: arrival sequence and keyed sequence.

Records that are retrieved using an **arrival sequence access path** will be retrieved in the same order in which they were added to the file. This is similar to processing sequential files. New records are physically stored at the end of the file. An arrival sequence access path is valid for both physical and logical files.

Records that are retrieved using a **keyed sequence access path** are retrieved based on the contents of one or more key fields in the record. This is similar to processing indexed or keyed files on other systems. A keyed sequence access path is updated whenever records are added, deleted, or updated, or when the contents of the key field are changed. This access path is valid for both physical and logical files.

If a file defines more than one record format, each record format may have different key fields. The default key for the file (for example, if no format is specified) will be the key fields that all record formats have in common. If there is no default key (for example, no common key fields), the first record in the file will always be returned on an input operation that does not specify the format.

Note: When your ILE C/C++ program opens a file, the default is to process the file with the access path that is used to create the file. If you specify `arrseq=N` (the default), the file is processed the way it was created. This means that if the file was created using a keyed sequence access path, your ILE C/C++ program processes the file by using a keyed sequence access path. If you specify `arrseq=Y`, the file is processed using arrival sequence. This means that even though the file was created using a keyed sequence access path, your ILE C/C++ program processes the file by using an arrival sequence access path.

Arranging Key Fields

Keyed sequence access paths can be ordered in ascending or descending sequence. When you describe a key field, the default is ascending sequence. If you are using Data Description Specifications (DDS) to create a keyed sequence file, the `DESCEND` DDS keyword can be used to specify that the key fields are to be arranged in descending sequence.

Duplicate Key Values

When a record has key fields whose contents are the same as another record's key fields in the same file, the file has records with duplicate key values. For example, if the record has two key fields *num* and *date*, duplicate key values occur when the contents of both *num* and *date* are the same in two or more records.

If you want an indication that your program is processing a record that contains a duplicate key value, specify `dupkey=y` on the call to `_Ropen()` that opens the file. If an I/O operation on a record is successful and a duplicate key value is found in that record, the `dup_key` flag in the `_RIOFB_T` structure is set. (The `_Rreadd()` function does not update this flag.)

Note: Using the `dupkey=y` option on the call to the `_Ropen()` function may cause your I/O operations to be slower.

You can avoid duplicate key values by specifying the keyword **UNIQUE** in the DDS file.

Deleted Records

When a database record is deleted, the physical record is marked as deleted but remains in the file. Deleted records can be overwritten by using the `_Rwrite()` function. Deleted records can be removed from a file by using the **RGZPFM** (Reorganize Physical File Member) command. They can also be reused on write operations by specifying the **REUSEDLT(*YES)** parameter on the **CRTPF** command.

Deleted records can occur in a file if the file has been initialized with deleted records using the **Initialize Physical File Member (INZPFM)** command. Once a record is deleted, it cannot be read.

Locking

The iSeries database has built-in record integrity. The system determines the lock conditions that are based on how your ILE C/C++ program opens the file. This table shows the valid open modes and the lock states that are associated with them:

Table 13. Lock States for Open Modes

Open Mode	Lock State
r, rb	shared for read (*SHRRD)
a, w, ab, wb, a+, r+, w+, ab+, rb+, wb+	shared for update (*SHRUPD)

You can change the lock state for a file by using the **Override Database File (OVRDBF)** command or the **Allocate Object (ALCOBJ)** command before you open the file. For example, your ILE C program can use the `system()` function to call the **ALCOBJ** command:

```
system("ALCOBJ OBJ((FILEA *FILE *EXCLRD))");
```

If a file is opened for update, the database locks any record read or positioned to provided the `__NO_LOCK` option is not specified. This means that the locked record cannot be locked to any other open data path, whether that open data path is opened by another program or even by the same program through another file pointer.

Successfully reading and locking another record releases the lock on a previously locked record. If the `__NO_LOCK` option is specified on any read then the lock on the previously locked record is not released. You can also release a lock on a record by using the `_Rrlslck()` function.

Sharing

If your application consists only of C and C++ modules, the preferred way to share a file is by opening the file in one program and passing the file pointer to the other programs. This eliminates the need to open the file more than once.

Sharing a file in the same job allows programs in that job to share the file's status, record position, and buffer. The **SHARE(*YES)** parameter on the create file, change file, and override the file commands allows an Open Data Path (ODP) to be shared

between two or more programs running in the same job. An **open data path** is the path through which all I/O operations for a file are performed.

You can share open data paths for streams processed a record at a time. You can also share open data paths for record files. You should not share the open data path for streams processed a character at a time, as unpredictable results can occur when you perform I/O operations.

Note:

- If you want to share a file between your C/C++ programs and programs that are written in other languages, you can do this by sharing an open data path.
- The first open of a shared file determines the open mode for the file (for example, whether it is open for INPUT, OUTPUT, UPDATE, and DELETE). If a subsequent open specifies an open mode that was not specified by the first open, the file will be opened the second time but the open mode will be ignored. For example, if the first open specifies an open mode of IO and the second open specifies IOUD, the file will be opened the second time with a mode of IO.

Null Capable Fields

The ILE C compiler allows you to process files with records that may contain fields that are considered to be null. You must specify `nullcap=Y` on the `_Ropen()` function. If a null-capable field is set to null, any data that is written into that field is not valid.

If a file is opened with `nullcap=Y`, the database provides input and output null maps as well as a key null map, if the file is keyed. The input and output null maps consist of one byte for each field in the current record format of the file. These null field maps are used to communicate between the database and your program to indicate which specific fields should be considered null.

The `_RFILE` structure defined in the `<recio.h>` file contains pointers to the input, output and key null field maps, and the lengths of these maps (`null_map_len` and `null_key_map_len`).

When you write to a database file, you specify which fields are null with a character '1'. If a field is not null you specify the character '0'. This is specified in the null field map pointed to by the `out_null_map` pointer. If the file does not contain any null capable fields, but has been opened with `nullcap=Y`, your program must set each field in the null field map to the character '0'. This must be done prior to writing any data to the file.

When you read from a database file, the corresponding byte in the null field map is indicated with a character '1' if the field is considered null. This is specified in the null field map pointed to by the `in_null_map` pointer.

The null key field map consists of one byte for each field in the key for the current record format. If you are reading a database file by key which has null fields, you must first indicate in the null key map pointed to by `null_key_map` which fields contain null. Specify character '1' for any field to be considered null, and character '0' for the other fields.

When the `_Rupdate()` function is called to update a file which has been opened to allow null field processing, the system input buffer is used. As a result, the database requires that an input null field map be provided through the `in_null_map` pointer. Prior to calling `_Rupdate()`, the user must clear and then set the input null field map (using the `in_null_map` pointer) according to the data which will be used to update the record.

You can use the `#pragma mapinc` directive to generate typedefs that correspond to the null field maps. You can cast the null field map pointers in the `_RFILE` structures to these types to manipulate these maps. Null field macros have also been provided in the `<recio.h>` file to assist users in clearing and setting the null field maps in their programs.

Opening Database and DDM Files as Record Files

To open an iSeries database file as a record file, use the `_Ropen()` function with one of the following modes:

- `rr` • `wr` • `ar`
- `rr+` • `wr+` • `ar+`

The valid keyword parameters for database and DDM files are:

- `arrseq` • `ccsid` • `riofb` • `vlr`
- `blkrcd` • `dupkey` • `secure` • `rtncode`
- `commit` • `nullcap` • `varparm`

Record Functions for Database and DDM Files

Use the following record functions to process database and DDM files:

- `_Rclose()` • `_Ropen()` • `_Rreadp()`
- `_Rcommit()` • `_Ropnfbk()` • `_Rr1slck()`
- `_Rdelete()` • `_Rreadd()` • `_Rrollbck()`
- `_Rfeod()` • `_Rreadf()` • `_Rupfb()`
- `_Rformat()` (multi-format • `_Rreadk()` • `_Rupdate()`
logical files) • `_Rreadl()` • `_Rwrit()`
- `_Riofbk()` • `_Rreadn()` • `_Rwritd()`
- `_Rlocate()`

I/O Considerations for DDM Files

DDM files may be accessed as program described files (specify the remote file name on the `RMTRFILE` parameter of the `CRTDDMF` command), or as externally described files (specify the remote DDS file name on the `RMTRFILE` parameter of the `CRTDDMF` command).

Opening Database and DDM Files as Binary Stream Files

To open an iSeries database file as a binary stream file for record-at-a-time processing, use the `fopen()` function with one of the following modes:

- `rb` • `wb` • `ab`

The valid keyword parameters for database and DDM files are:

- blksize
- type
- arrseq
- lrecl
- commit
- ccsid

If you specify a database or a DDM file the parameter type must be "record".

Note: The physical database files that are created when the database file does not exist (where the open mode is wb or ab) are equivalent to specifying the following CL command:

```
CRTPF FILE(filename) RCDLEN(lrecl)
```

Records in this file are created with a record length that is based on the keyword parameter lrecl.

The only way to create a DDM file is to use the Create DDM File (CRTDDMF) command. If you use the fopen() function with a mode of wb or ab and the DDM file exists on the source system, but the database file does not exist on the remote system, a physical database file is created on the remote system. If the DDM file does not exist on the source system, a physical database file is created on the source system.

I/O Considerations for Binary Stream Database and DDM Files

If the database file contains deleted records, the deleted records are skipped by all binary stream I/O functions.

Binary stream record-at-a-time files cannot be processed by key. As well, they can only be opened with the rb, wb, and ab modes.

Binary Stream Functions for Database and DDM Files

Use one of the following binary stream functions to process database files and DDM files one record at time:

- fclose()
- fread()
- fwrite()
- fopen()
- freopen()

Processing a Database Record File in Arrival Sequence

You can copy data from one file to another file by using an arrival sequence access path. The records are accessed in the file in the same order in which they are added to the file.

Example

The following example copies data from the input file T1520ASI to the output file T1520ASO by using the same order in which they are added to the file T1520ASI. The _Rreadn() and _Rwrite() functions are used.

1. To create a physical file T1520ASI for the input, type:
CRTPF FILE(MYLIB/T1520ASI) RCDLEN(300)
2. Type the following sample data into T1520ASI:

```
joe 5
fred 6
wilma 7
```

3. To create a physical file T1520ASO for the output, type:
CRTPF FILE(MYLIB/T1520ASO) RCDLEN(300)
4. To create the program T1520ASP using the source shown below, type:
CRTBNDC PGM(MYLIB/T1520ASP) SCRFILE(QCLE/QACSRC)

```
/* This program illustrates how to copy records from one file to      */
/* another file, using the _Readn(), and _Rwrite() functions.          */

#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

#define _RCDLEN 300

int main(void)
{
    _RFILE    *in;
    _RFILE    *out;
    _RIOFB_T  *fb;
    char      record[_RCDLEN];

    /* Open the input file for processing in arrival sequence.          */

    if ( (in = _Ropen("LIBL/T1520ASI", "rr, arrseq=Y")) == NULL )
    {
        printf("Open failed for input file\n");
        exit(1);
    };

    /* Open the output file.                                             */

    if ( (out = _Ropen("LIBL/T1520ASO", "wr")) == NULL )
    {
        printf("Open failed for output file\n");
        exit(2);
    };

    /* Copy the file until the end-of-file condition occurs.            */

    fb = _Readn(in, record, _RCDLEN, __DFT);
    while ( fb->num_bytes != EOF )
    {
        _Rwrite(out, record, _RCDLEN);
        fb = _Readn(in, record, _RCDLEN, __DFT);
    };

    _Rclose(in);
    _Rclose(out);
}
```

Figure 88. T1520ASP — ILE C Source to Process a Database Record File in Arrival Sequence

This program uses the `_Ropen()` function to open the input file T1520ASI to access the records in the same order that they are added. The `_Ropen()` function

also opens the output file T1520ASO. The `_Rread()` function reads the records in the file T1520ASI. The `_Rwrite()` function writes them to the file T1520ASO.

5. To run the program type:

```
CALL PGM(MYLIB/T1520ASP)
```

The physical file T1520ASO contains the following data:

```
joe 5
fred 6
wilma 7
```

Processing a Database Record File in Keyed Sequence

You can update a record file by using a keyed sequence access path. The records are arranged based on the contents of one or more key fields in the record.

Example

The following example updates data in the record file T1520DD3 by using the key field SERIALNUM. The `_Rupdate()` function is used.

1. Type:

```
CRTPF FILE(MYLIB/T1520DD3) SRCFILE(QCLE/QADDSSRC)
```

To create the physical file T1520DD3 that uses the following DDS source:

```
A          R PURCHASE
A          ITEMNAME      10
A          SERIALNUM     10
A          K SERIALNUM
```

Figure 89. T1520DD3 — DDS Source for Database Records

2. Type the following sample data into T1520DD3:

```
orange 1000222200
grape  1000222010
apple  1000222030
cherry 1000222020
```

Although you enter the data as shown, the file T1520DD3 is accessed by the program T1520KSP in keyed sequence. Therefore the program T1520KSP reads the file T1520DD3 in the following sequence:

```
grape 1000222010
cherry 1000222020
apple 1000222030
orange 1000222200
```

3. Type:

```
CRTBNDC PGM(MYLIB/T1520KSP) SRCFILE(QCLE/QACSRC)
```

To create the program T1520KSP, using the following source:

```

/* This program illustrates how to update a record in a file using */
/* the _Rupdate() function. */

#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    _RFILE    *in;
    char      new_purchase[21] = "PEAR      1002022244";

    /* Open the file for processing in keyed sequence. File is created */
    /* with the default access path. */

    if ( (in = _Ropen("LIBL/T1520DD3", "rr+")) == NULL )
    {
        printf("Open failed\n");
        exit(1);
    };
    /* Update the first record in the keyed sequence. The function */
    /* _Rlocate locks the record. */

    _Rlocate(in, NULL, 0, __FIRST);
    _Rupdate(in, new_purchase, 20);
    /* Force the end of data. */

    _Rfeod(in);

    _Rclose(in);
}

```

Figure 90. T1520KSP — ILE C Source to Process a Database Record File in Keyed Sequence

This program uses the `_Ropen()` function to open the record file T1520DD3. The default access path which is the keyed sequence access path is used to create the file T1520DD3. The `_Rlocate()` function locks the first record in the keyed sequence. The `_Rupdate()` function updates the record that is locked by `_Rlocate()` to PEAR 1002022244. (The first record becomes the second record in the keyed sequence access path because the key has changed.)

4. To run the program T1520KSP, type:

```
CALL PGM(MYLIB/T1520KSP)
```

Since grape is the first record in the keyed sequence, it is updated, and the data file T1520DD3 is as follows:

```

orange  1000222200
PEAR    1002022244
apple   1000222030
cherry  1000222020

```

Processing a Database Record File Using Record Input and Output Functions

You can read and print records from a data file.

Example

The following example uses the `_Ropen()`, `_Rreadl()`, `_Rreadp()`, `_Rreads()`, `_Rreadd()`, `_Rreadf()`, `_Rrslck()`, `_Rdelete()`, `_Ropnfbk()`, and `_Rclose()` record I/O functions. The program `T1520REC` reads and prints records from the data file `T1520DD4`.

1. To create the physical file `T1520DD4` that uses the following DDS:

```

A          R PURCHASE
A          ITEMNAME      10
A          SERIALNUM     10
A          K SERIALNUM

```

Figure 91. T1520DD4 — DDS Source for Database Records

Type:

```
CRTPF FILE(MYLIB/T1520DD4) SRCFILE(QCLE/QADDSSRC)
```

2. Type the following sample data into `T1520DD4`:

```

orange  1000222200
grape   1000222010
apple   1000222030
cherry  1000222020

```

3. Type:

```
CRTBND CPGM(MYLIB/T1520REC) SRCFILE(QCLE/QACSRC).
```

To create the program `T1520REC` that uses the following source:

```

/* This program illustrates how to use the _Rreadp(), _Rreads(),      */
/* _Rreadd(), _Rreadf(), _Rreadn(),                                  */
/* _Ropnfbk(), _Rdelete, and _Rrslck() functions.                    */

#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    char      buf[21];
    _RFILE    *fp;
    _XXOPFB_T *opfb;
    /* Open the file for processing in arrival sequence.              */

    if (( fp = _Ropen ( "*/LIBL/T1520DD4", "rr+", arrseq=Y" )) == NULL )
    {
        printf ( "Open failed\n" );
        exit ( 1 );
    };
    /* Get the library and file names of the file opened.            */

    opfb = _Ropnfbk ( fp );
    printf ( "Library: %10.10s\nFile:   %10.10s\n",
            opfb->library_name,
            opfb->file_name);

```

Figure 92. T1520REC — ILE C Source to Process a Database File Using Record I/O Functions (Part 1 of 2)

```

/* Get the last record. */

    _Rreadl ( fp, NULL, 20, __DFT );
    printf ( "Fourth record: %10.10s\n", *(fp->in_buf) );

/* Get the third record. */

    _Rreadp ( fp, NULL, 20, __DFT );
    printf ( "Third record: %10.10s\n", *(fp->in_buf) );

/* Release lock on the record so another function can access it. */

    _Rrslck ( fp );
/* Read the same record. */

    _Rreads ( fp, NULL, 20, __DFT );
    printf ( "Same record: %10.10s\n", *(fp->in_buf) );

/* Get the second record without locking it. */

    _Rreadd ( fp, NULL, 20, __NO_LOCK, 2 );
    printf ( "Second record: %10.10s\n", *(fp->in_buf) );

/* Get the first record. */

    _Rreadf ( fp, NULL, 20, __DFT );
    printf ( "First record: %10.10s\n", *(fp->in_buf) );

/* Delete the second record. */

    _Rreadn ( fp, NULL, 20, __DFT );
    _Rdelete ( fp );

/* Read all records after deletion. */

    _Rreadf ( fp, NULL, 20, __DFT );
    printf ( "First record after deletion: %10.10s\n", *(fp->in_buf));
    _Rreadn ( fp, NULL, 20, __DFT );
    printf ( "Second record after deletion: %10.10s\n", *(fp->in_buf));
    _Rreadn ( fp, NULL, 20, __DFT );
    printf ( "Third record after deletion: %10.10s\n", *(fp->in_buf));

    _Rclose ( fp );
}

```

Figure 92. T1520REC — ILE C Source to Process a Database File Using Record I/O Functions (Part 2 of 2)

The `_Ropen()` function opens the file T1520DD4. The `_Ropnfbk()` function gets the library name MYLIB and file name T1520DD4. The `_Rreadl()` function reads the fourth record "cherry 1000222020". The `_Rreadp()` function reads the third record "apple 1000222030". The `_Rrslck()` function releases the lock on this record so that `_Rreads()` can read it again. The `_Rreadd()` function reads the second record "grape 1000222010" without locking it. The `_Rreadf()` function reads the first record "orange 100022200". The `_Rdelete()` function deletes the second record. All records are then read and printed.

4. To run the program T1520REC, type:
CALL PGM(MYLIB/T1520REC)

The output is as follows:

```
Library: MYLIB
File: T1520DD4
Fourth record: cherry
Third record: apple
Same record: apple
Second record: grape
First record: orange
First record after deletion: orange
Second record after deletion: apple
Third record after deletion: cherry
Press ENTER to end terminal session.
```

5. The physical file T1520DD4 contains the data that is shown:

```
ORANGE 1000222200
APPLE 1000222030
CHERRY 1000222020
```

Grouping File Operations Using Commitment Control

Commitment control is a means of grouping file operations as a single unit so that you can synchronize changes to database files in a single job.

Before you can start commitment control, you must ensure that all the database files you want processed as one unit are in one commitment control environment. All the files within this environment must be journaled to the same journal. Use the CL commands Create Journal Receiver (CRTJRNRCV), Create Journal (CRTJRN) and Start Journal Physical File (STRJRNPF) to prepare for the journaling environment.

Once the journaling environment is established, enter the following commands:

- Start Commitment Control (STRCMTCTL)
- CALL *program-name*
- End Commitment Control (ENDCMTCTL).

You can use commitment control to define and process several changes to database files as a single transaction.

Example

The following example uses commitment control. Purchase orders are entered and logged in two files, T1520DD5 for daily transactions, and T1520DD6 for monthly transactions. Journal entries that reflect the changes that are made to T1520DD5 and T1520DD6 are kept in the journal JRN.

1. To create the physical file T1520DD5 using the DDS source shown below, type:
CRTPF FILE(QTEMP/T1520DD5) SRCFILE(QCLE/QADDSSRC)

```
A          R PURCHASE
A          ITEMNAME    30
A          SERIALNUM   10
```

Figure 93. T1520DD5 — DDS Source for Daily Transactions

2. To create the physical file T1520DD6 using the DDS source shown below, type:
CRTPF FILE(QTEMP/T1520DD6) SRCFILE(QCLE/QADDSSRC)

```

A          R PURCHASE
A          ITEMNAME      30
A          SERIALNUM      10

```

Figure 94. T1520DD6 — DDS Source for Monthly Transactions

3. To create the physical file NFTOBJ for notification text, type:
CRTPF FILE(MYLIB/NFTOBJ) RCDLEN(19)

Notification text is sent to the file NFTOBJ when the ILE C program T1520COM that uses commitment control is run.

4. To create the display file T1520DD7 using the DDS source shown below, type:
CRTDSPF FILE(QTEMP/T1520DD7) SRCFILE(QCLE/QADSSRC)

```

A          DSPSIZ(24 80 *DS3)
A          REF(QTEMP/T1520DD5)
A          INDARA
A          CF03(03 'EXIT ORDER ENTRY')
A          R PURCHASE
A          3 32'PURCHASE ORDER FORM'
A          DSPATR(UL)
A          DSPATR(HI)
A          10 20'ITEM NAME      :'
A          DSPATR(HI)
A          12 20'SERIAL NUMBER :'
A          DSPATR(HI)
A          ITEMNAME R      I 10 37
A          SERIALNUM R     I 12 37
A          23 34'F3 - Exit'
A          DSPATR(HI)
A          R ERROR
A          6 28'ERROR: Write failed'
A          DSPATR(BL)
A          DSPATR(UL)
A          DSPATR(HI)
A          10 26'Purchase order entry ended'

```

Figure 95. T1520DD7 — DDS Source for a Purchase Order Display

5. To create the journal receiver JRNRCV, type:
CRTJRNRCV JRNRCV(MYLIB/JRNRCV)

Journal entries are placed in JRNRCV when the application is run.

6. To create the journal JRN and attach the journal receiver JRNRCV to it, type:
CRTJRN JRN(MYLIB/JRN) JRNRCV(MYLIB/JRNRCV)

7. To start journaling the changes that are made to T1520DD5 and T1520DD6 in the journal JRN, type:

```

STRJRNPF FILE(QTEMP/T1520DD5 QTEMP/T1520DD6) JRN(MYLIB/JRN)
IMAGES(*BOTH)

```

8. To create the program T1520COM using the program source shown below, type:

```

CRTBND C PGM(MYLIB/T1520COM) SRCFILE(QCLE/QACSRC)

```

```

/* This program illustrates how to use commitment control using the */
/* _Rcommit() function and to rollback a transaction using the */
/* _Rollbck() function. */

#include <stdio.h>
#include <recio.h>
#include <stdlib.h>
#include <string.h>

#define PF03 2
#define IND_OFF '0'
#define IND_ON '1'
int main(void)
{
    char    buf[40];
    int     rc = 1;
    _SYSindara ind_area;
    _RFILE  *purf;
    _RFILE  *dailyf;
    _RFILE  *monthlyf;

/* Open purchase display file, daily transaction file and monthly */
/* transaction file. */
    if ( ( purf = _Ropen ( "*LIBL/T1520DD7", "ar+,indicators=y" ) ) == NULL )
    {
        printf ( "Display file did not open.\n" );
        exit ( 1 );
    }
    if ( ( dailyf = _Ropen ( "*LIBL/T1520DD5", "wr,commit=y" ) ) == NULL )
    {
        printf ( "Daily transaction file did not open.\n" );
        exit ( 2 );
    }

    if ( ( monthlyf = _Ropen ( "*LIBL/T1520DD6", "ar,commit=y" ) ) == NULL )
    {
        printf ( "Monthly transaction file did not open.\n" );
        exit ( 3 );
    }

/* The associate separate indicator area with the purchase file. */
    _Rindara ( purf, ind_area );

/* Select the purchase record format. */
    _Rformat ( purf, "PURCHASE" );

/* Invite the user to enter a purchase transaction. */
/* The _Rwrite function writes the purchase display. */

    _Rwrite ( purf, "", 0 );
    _Rreadn ( purf, buf, sizeof(buf), __DFT );
/* While the user is entering transactions, update daily and */
/* monthly transaction files. */

```

Figure 96. T1520COM — ILE C Source to Group File Operations Using Commitment Control (Part 1 of 2)

```

while ( rc && ind_area[PF03] == IND_OFF )
{
    rc = (( _Rwrite ( dailyf, buf, sizeof(buf) ))->num_bytes );
    rc = rc && ( _Rwrite ( monthlyf, buf, sizeof(buf) ))->num_bytes;

    /* If the databases were updated, then commit transaction.          */
    /* Otherwise, rollback the transaction and indicate to the          */
    /* user that an error has occurred and end the application.          */

    if ( rc )
    {
        _Rcommit ( "Transaction complete" );
    }
    else
    {
        _Rrollbck ( );
        _Rformat ( purf, "ERROR" );
    }
    _Rwrite ( purf, "", 0 );
    _Rreadn ( purf, buf, sizeof(buf), __DFT );
}
}

```

Figure 96. T1520COM — ILE C Source to Group File Operations Using Commitment Control (Part 2 of 2)

The `_Ropen()` function opens the purchase display file, the daily transaction file, and the monthly transaction file. The `_Rindara()` function identifies a separate indicator area for the purchase file. The `_Rformat()` function selects the purchase record format defined in T1520DD7. The `_Rwrite()` function writes the purchase order display. Data that is entered updates the daily and monthly transaction files T1520DD5 and T1520DD6. The transactions are committed to these database files that use the `_Rcommit()` function.

9. To run program T1520COM under commitment control, type:
`STRCMTCTL LCKLVL(*CHG) NFYOBJ(MYLIB/NFTOBJ (*FILE)) CMTSCOPE(*JOB)`
`CALL PGM(MYLIB/T1520COM)`

The display appears as follows:

PURCHASE ORDER FORM

ITEM NAME :

SERIAL NUMBER :

F3 - Exit

10. Type in the following sample data into the Purchase Order Form display:

TABLE	12345
BENCH	43623
CHAIR	62513

After an item and serial number are entered, T1520DD5 and T1520DD6 files are updated. The sample data shows that the contents of the daily transaction file T1520DD5 file after three purchase order items are entered.

11. To end commitment control, type:

ENDCMTCTL

The journal JRN contains entries that correspond to changes that are made to T1520DD5 and T1520DD6.

Blocking Records

You can use record blocking to improve the performance of I/O operations on files that are opened for input or output only. Specify the `blksize=value` parameter on a call to the `fopen()` function or the `blkrcd=y` on a call to the `_Ropen()` function to turn on record blocking. In some situations, the operating system will return only one record in the block when processing a file. In these cases there is no performance gain.

You can turn off record blocking without changing your program by specifying `SEQONLY(*YES)` on the `OVRDBF` command.

Note: When record blocking is in effect, the I/O feedback structure is only updated when a block of records is transferred between your program and the system.

Chapter 12. Using Device Files in Your Programs

This chapter describes how to:

- Specify indicators as part of the file buffer to be read or written
- Return indicators in a separate indicator area
- Establish a default program device
- Change a default program device
- Use the `_Riobufk()` function to obtain feedback information
- Write source statements to a tape file
- Write source statements to a diskette file
- Use save files

OS/400 Feedback Areas for all Device Files

To access the device attributes feedback area, use the `_Rdevatr()` function. To use stream files (type=record) with record I/O functions, you must cast the FILE pointer to an `_RFILE` pointer.

Display Files, Intersystem Communication Files and Printer Files

Separate indicator areas, and major and minor return codes apply to display, ICF, and printer files.

I/O Considerations

Indicators allow information to be passed from a program to the system or from the system to a program. Display, ICF, and printer files can make use of indicators. Indicators are boolean data items that can contain a value of either 1 or 0 (character). There are two types of indicators:

Option Indicators pass information from a program to the system. For example, they can control which fields in a record can be displayed.

Response Indicators pass information from the system to an application when an input request finishes. For example, they can be used to inform the program which function keys were pressed by the workstation user.

To use indicators, the display, ICF, and printer files must be defined as an externally described file. The data description specification (DDS) for the externally described display file must contain a one-character INDICATOR field for each indicator. Indicators are either in the records read or written by the program (the indicators are in the file buffer) or in a separate indicator area.

Separate Indicator Areas

If you specify the INDARA keyword in the DDS, the indicators for the display, ICF, and printer files are returned in a separate indicator area. An **indicator area** is a 99-element character array with indices from 0-98.

The display, ICF, and printer files must be opened with the keyword indicators=y for the indicators to be specified in a separate indicator area. Use the `_Rindara()` function to identify the separate indicator buffer associated with the file.

If you do not specify the **INDARA** keyword in the DDS, the indicators for the display, ICF, or printer file will be specified in the record buffer. The number and order of the indicators that are defined in the DDS determine the number and order of the indicators in the record buffer. Indicators are always positioned first in the record buffer. The `in_buf` and `out_buf` pointers in the `_RFILE` structure point to the input and output record buffers for a file.

Major and Minor Return Codes

Major and minor return codes are used to report certain status information for display, ICF, and printer files. After a read (`_Rreadindv()` or `_Rreadn()`) or write (`_Rwrite()`) operation, the `sysparm` field in the `_RIOFB_T` structure points to the major/minor return code for the display, ICF or printer files. The header file `<recio.h>` declares the `_RIOFB_T` structure.

The *Application Display Programming* manual describes major and minor return codes and their meanings for display files. The *Printer Device Programming* manual describes major and minor return codes and their meanings for printer files.

Your program should test the return code after each I/O operation and define any error handling operations that are based on the major and minor return codes. If the major return code is 00, the operation completed successfully. If an error occurs with a display, ICF, or printer file your program should handle it as it occurs.

Display Files and Subfiles

A **display** file is used to define the format of the information that you wish to present on a display. It also defines how that information is processed by the system on its way to and from the display.

A **subfile** is a display file that contains a group of records with the same record format that can be accessed by relative record number. The records of a subfile can be displayed on a display station. The system sends the entire group of records to the display in a single operation and receives the group from the display in another operation. The object type for both is `*FILE`.

To work with externally described display files use one of:

- DDS through the Code/400 editor or the SEU.
- Screen Design Aid (SDA) or DSU What-You-See-Is-What-You-Get (WYSIWYG) tools.

I/O Considerations for Display Files

- An ILE C/C++ program can process display files as program described files or as externally described files:
 - For program described display files, specify all formatting and control information in the ILE C/C++ program that uses the file. To create a program described display file, specify `SRCFILE(*NONE)` on the `CRTDSPF` command.
 - For externally described display files, specify all formatting and control information using DDS to describe the layout of the display. To create an externally described display file, specify the name of the member that contains the DDS source on the `SRCFILE` parameter of the `CRTDSPF` command.
- If you are using a user-defined data stream (UDDS), hexadecimal 3F (X'3F') blanks the display until the next display attribute. If any CCSID conversion takes place and a character cannot be mapped to the corresponding character in

another code page, the character is mapped to hexadecimal 3F. This will blank the screen until the next display attribute. See Chapter 18, “Internationalizing Your Program” on page 403 for information on CCSIDs.

- The concept of clearing a file or opening a file using append mode does not apply to display files.

I/O Considerations for Subfiles

- The input typedef for record format subfiles will contain fields with the usage of I, O, B, and H. Input typedef for control record format subfiles will contain fields with the usage of I, B, and H.
- To use a subfile, you initialize it, for example, by reading records from a database file and writing them to a subfile. You must write them using `_Rwrited`.

Opening Display Files and Subfiles as Binary Stream Files

To open an iSeries display file or subfile as a binary stream file for record-at-a-time processing, use the `fopen()` function with one of the following modes:

- `rb`
- `ab+`
- `wb` and `ab`

The CRTDSPF command is the only way to create a display file. If you use the `fopen()` function and the display file does not exist, a physical database file is created.

The valid keyword parameters are:

- `type`
- `lrecl`
- `indicators`

I/O Considerations for Binary Stream Subfiles

Only message subfiles are supported for binary stream subfiles.

Program Devices for Binary Stream Display Files

The program device that is associated with display files is a workstation. You establish the default device by implicitly acquiring it using the `fopen()` function.

Binary Stream Functions for Display Files and Subfiles

Use the following binary stream functions to process display files and subfiles:

- `fclose()`
- `fread()`
- `fwrite()`
- `fopen()`
- `freopen()`

Open Display Files as Record Files

To open an iSeries display file or subfile as a record file, use the `_Ropen()` function with one of the following modes:

- `rr`
- `wr` and `ar`
- `ar+`
- `rr+` and `wr+`

The valid keyword parameters are:

- `lrecl`
- `indicators`
- `secure`
- `riofb`

I/O Considerations for Record Display Files

The program device that is associated with display files is a workstation. You establish the default device by implicitly acquiring it using the `_Ropen()` function. The implicitly acquired program device is determined by the `DEV` parameter on the `CRTDSPF`, `CHGDSPF`, or `OVRDSPF` commands. If `*REQUESTER` is specified on the `DEV` parameter, then the program device from which the program was called is implicitly acquired. It becomes the default program device for I/O operations to the display file.

If `*NONE` is specified on the `DEV` parameter of the `CRTDSPF`, `CHGDSPF`, or `OVRDSPF` commands, you must explicitly acquire the program device with the `_Racquire()` function. The explicitly acquired program device now becomes the default device for subsequent I/O operations to the device file.

You can change the default program device in the following ways:

- Use the `Racquire()` function to explicitly acquire another program device. The device that is just acquired becomes the current program device.
- Use the `_Rpgmdev()` function to change the current program device that is associated with a file to a previously-acquired device. This program device can be used for subsequent input and output operations to the file.
- The actual program device that is read becomes the default device if you read from an invited device using the `_Rreadindv()` function.
- Use the `_Rrelease()` function to release a device from the file. When you release the device, it is no longer available for I/O operations.

I/O Considerations for Record Subfiles

I/O operations to the subfile record format do not cause data to appear on the display. You must read or write the subfile control record format to transfer data to or from the display. Use the `_Rformat()` function to distinguish between subfile record formats and subfile control formats. If the format you specify with the `_Rformat()` function refers to a subfile record format, no data is transferred to or from the display.

To read the next changed subfile record, use the `_Rreadnc()` function. This function searches for the next changed record from the current position in the file. If this is the first read operation, the first changed record in the subfile is read. If the end-of-file is reached before finding a changed record, EOF is returned in the `num_bytes` field of the `_RIOFB_T` structure.

Record Functions for Display Files and Subfiles

Use the following record functions to process display files and subfiles:

- | | | |
|----------------------------|--------------------------------------|--------------------------------------|
| • <code>_Racquire()</code> | • <code>_Ropen()</code> | • <code>_Rrelease()</code> |
| • <code>_Rclose()</code> | • <code>_Ropnfbk()</code> | • <code>_Rupdate()</code> (subfiles) |
| • <code>_Rdevatr()</code> | • <code>_Rpgmdev()</code> | • <code>_Rupfb()</code> |
| • <code>_Rfeod()</code> | • <code>_Rreadd()</code> (subfiles) | • <code>_Rwrite()</code> |
| • <code>_Rformat()</code> | • <code>_Rreadindv()</code> | • <code>_Rwrited()</code> (subfiles) |
| • <code>_Rindara()</code> | • <code>_Rreadn()</code> | • <code>_Rwriterd()</code> |
| • <code>_Riofbk()</code> | • <code>_Rreadnc()</code> (subfiles) | • <code>_Rwrread()</code> |

Specifying Indicators as Part of the File Buffer

You can specify indicators in records read or written by a program. Indicators can pass information from a program to the system or from the system to a program.

Example

The following example shows how to specify an indicator in a record that is read by program T1520ID1. The indicator is placed in the file buffer of an externally described file. The DDS for the externally described file contains one character indicator field.

1. To create the display file T1520DD9 using the DDS source shown below, type:
CRTDSPF FILE(MYLIB/T1520DD9) SRCFILE(QCLE/QADDSSRC)

```
A          R PHONE
A
A          CF03(03 'EXIT')
A          1 35'PHONE BOOK'
A          DSPATR(HI)
A          7 28'Name:'
A          NAME          11A I 7 34
A          9 25'Address:'
A          ADDRESS      20A I 9 34
A          11 25'Phone #:'
A          PHONE_NUM    8A I 11 34
A          23 34'F3 - EXIT'
A          DSPATR(HI)
```

Figure 97. T1520DD9 — DDS Source for a Phone Book Display

2. To create the program T1520ID1 using the program source shown below, type:
CRTBNDC PGM(MYLIB/T1520ID1) SRCFILE(QCLE/QACSRC)

```
/* This program uses a response indicator to inform the program      */
/* that F3 was pressed by a user. The response indicator is returned */
/* in part of the file buffer.                                       */
#include <stdio.h>
#include <recio.h>
#include <stdlib.h>

typedef struct{
    char in03;
    char name[11];
    char address[20];
    char phone_num[8];
}info;

#define IND_ON '1'
```

Figure 98. T1520ID1 — ILE C Source to Specify Indicators as Part of the File Buffer (Part 1 of 2)

```

int main(void)
{
    _RFILE    *fp;
    _RIOFB_T  *rfb;
    info      phone_list;

    if (( fp = _Ropen ( "*LIBL/T1520DD9", "ar+" )) == NULL )
    {
        printf ( "display file open failed\n" );
        exit ( 1 );
    }

    _Rformat ( fp, "PHONE" );
    rfb = _Rwrite ( fp, "", 0);
    rfb = _Rreadn ( fp, &phone_list, sizeof(phone_list), __DFT );
    if ( phone_list.in03 == IND_ON )
    {
        printf ( "user pressed F3\n" );
    }
    _Rclose ( fp );
}

```

Figure 98. T1520ID1 — ILE C Source to Specify Indicators as Part of the File Buffer (Part 2 of 2)

This program uses a response indicator IND_ON '1' to inform the ILE C program T1520ID1 that a user pressed F3.

3. To run the program T1520ID1, type:

```
CALL PGM(MYLIB/T1520ID1)
```

The output is as follows:

```

                                PHONE BOOK
                                Name:
                                Address:
                                Phone #:
                                F3 - EXIT

```

Specifying Indicators in a Separate Indicator Area

You can specify indicators in records to be read or written by a program in a separate indicator area using the INDARA keyword in DDS.

Example

The following example illustrates how indicators are returned in a separate indicator area. The INDARA keyword that is specified in the DDS means that the indicator for the display is returned to a separate indicator area.

1. To create the display file T1520DD0 using the DDS source shown below, type:

```
CRTDSPF FILE(MYLIB/T1520DD0) SRCFILE(QCLE/QADDSSRC)
```

```

A                                INDARA
A      R PHONE
A                                CF03(03 'EXIT')
A                                1 35'PHONE BOOK'
A                                DSPATR(HI)
A                                7 28'Name:'
A      NAME      11A I 7 34
A                                9 25'Address:'
A      ADDRESS   20A I 9 34
A                                11 25'Phone #:'
A      PHONE_NUM 8A I 11 34
A                                23 34'F3 - EXIT'
A                                DSPATR(HI)

```

Figure 99. T1520DD0 — DDS Source for a Phone Book Display

2. To create the program T1520ID2 using the source shown below, type:

```
CRTBNDC PGM(MYLIB/T1520ID2) SRCFILE(QCLE/QACSRC)
```

```

/* This program uses response indicators to inform the program that */
/* F3 was pressed by a user to indicate that an input request      */
/* finished. The response indicators are returned in a separate    */
/* indicator area.                                                */
#include <stdio.h>
#include <recio.h>
#include <stdlib.h>
typedef struct{
    char name[11];
    char address[20];
    char phone_num[8];
}info;
#define IND_ON '1'
#define F3      2

int main(void)
{
    _RFILE      *fp;
    _RIOFB_T     *rfb;
    info         phone_list;
    _SYSindara indicator_area;
    if (( fp = _Ropen ( "*LIBL/T1520DD0", "ar+ indicators=y" )) == NULL )
    {
        printf ( "display file open failed\n" );
        exit ( 1 );
    }
    _Rindara ( fp, indicator_area );
    _Rformat ( fp, "PHONE" );
    rfb = _Rwrite ( fp, "", 0 );
    rfb = _Rreadn ( fp, &phone_list, sizeof(phone_list), __DFT );
    if ( indicator_area[F3] == IND_ON )
    {
        printf ( "user pressed F3\n" );
    }
    _Rclose ( fp );
}

```

Figure 100. T1520ID2 — ILE C Source to Specify Indicators in a Separate Indicator Area

This program uses response indicators IND_ON '1' and F3 2 to inform the ILE C program T1520ID2 that a user pressed F3. The _Rindara() function accesses the separate indicator buffer indicator_area associated with the externally described file T1520DD0. The display file T1520DD0 is opened with the keyword indicators=yes to return the indicator to a separate area.

3. To run the program T1520ID2, type:

```
CALL PGM(MYLIB/T1520ID2)
```

The output is as follows:

PHONE BOOK

Name:

Address:

Phone #:

F3 - EXIT

Establishing the Default Program Device

You can establish the default device for display and ICF files.

Example

The following example illustrates how to explicitly establish a default program device for a display file using the _Racquire() function.

Note: To run this example you must use a display device that is defined on your system in place of DEVICE2.

1. To create the display file T1520DDD using the DDS shown below, type:

```
CRTDSPF FILE(MYLIB/T1520DDD) SRCFILE(QCLE/QADDSSRC) MAXDEV(2)
```

```

A                                DSPSIZ(24 80 *DS3)
A      R EXAMPLE
A      OUTPUT      5A  0  5 20
A      INPUT      20A  I  7 20
A                                5 10'OUTPUT:'
A                                7 10'INPUT:'

```

Figure 101. T1520DDD — DDS Source for an I/O Display

2. To override the file STDOUT with the printer file QPRINT, type:
OVRPRTF FILE(STDOUT) TOFILE(QPRINT)
3. To create the program T1520DEV using the source shown below, type:
CRTBNDC PGM(MYLIB/T1520DEV) SRCFILE(QCLE/QACSRC)

```

/* This program establishes a default device using the _Racquire */
/* function. */

#include <stdio.h>
#include <recio.h>
#include <signal.h>
#include <stdlib.h>

void handler ( int );

int main(void)
{
    _RFILE *fp;
    _RIOFB_T *rfb;
    char buf[21];

    signal (SIGALL, handler );

    if (( fp = _Ropen ( "*/LIBL/T1520DDD","ar+" )) == NULL )
    {
        printf ( "Could not open the display file\n" );
        exit ( 1 );
    }
    _Racquire ( fp,"DEVICE2" );          /* Acquire the device. */
                                         /* DEVICE2 is now the */
                                         /* default program device. */
    _Rformat ( fp,"EXAMPLE" );          /* Select the record */
                                         /* format. */
    _Rwrite ( fp, "Hello", 5 );          /* Write to the default */
                                         /* program device. */

    rfb = _Rreadn ( fp, buf, 20, __DFT ); /* Read from the default */
                                         /* program device. */

    buf[rfb -> num_bytes] = '\0';

    printf ( "Response from device : %s\n", buf );

    _Rrelease ( fp, "DEVICE2" );
    _Rclose ( fp );
}

void handler ( int sig )
{
    printf ( "message = %7.7s\n", _EXCP_MSGID );
    printf ( "program continues \n" );
    signal ( SIGALL, handler );
}

```

Figure 102. T1520DEV — ILE C Source to Establish a Default Device

The `_Racquire()` function explicitly acquires the program device `DEVICE2`. `DEVICE2` is the current program device. The `_Rformat()` function selects the record format `EXAMPLE`. The `_Rwrite()` function writes data to the default device. The `_Rreadn()` function reads the string from the current program device `DEVICE2`.

4. To run the program `T1520DEV`, type:
`CALL PGM(MYLIB/T1520DEV)`

The output is as follows:

```
OUTPUT:  Hello
INPUT:   _____
```

5. Type GOOD MORNING on the input line and press Enter.

The file QPRINT contains:

Response from device : GOOD MORNING

Changing the Default Program Device

You can change the default device for a device file.

Example

The following example illustrates how to change the default program device using the `_Rpgmdev()` function.

Note: To run this example you must use two display devices that are defined on your system in place of DEVICE1 and DEVICE2.

1. To create the display file T1520DDE using the DDS shown below, type:

```
CRTDSPF FILE(MYLIB/T1520DDE) SRCFILE(QCLE/QADDSSRC) MAXDEV(2)
```

```
A                                DSPSIZ(24 80 *DS3)
A                                INVITE
A      R FORMAT1
A                                9 13'Name:'
A      NAME                20A  I  9 20
A                                11 10'Address:'
A      ADDRESS            25A  I 11 20
A      R FORMAT2
A                                9 13'Name:'
A      NAME                8A  I  9 20
A                                11 10'Password:'
A      PASSWORD          10A  I 11 20
```

Figure 103. T1520DDE — DDS Source for Name and Password Display

2. To override the file STDOUT with the printer file QPRINT, type:
`OVPRPTF FILE(STDOUT) TOFILE(QPRINT)`
3. To create the program T1520CDV using the source shown below, type:
`CRTBNDC PGM(MYLIB/T1520CDV) SRCFILE(QCLE/QACSRC)`


```

/* This program illustrates how to change a default program device. */
/* using the _Racquire(), _Rpgmdev(), _Rreadindv() and _Rrelease() */
/* functions. */

#include <stdio.h>
#include <recio.h>
#include <string.h>
#include <stdlib.h>

typedef struct{
    char name[20];
    char address[25];
}format1;
typedef struct{
    char name[8];
    char password[10];
}format2 ;

typedef union{
    format1 fmt1;
    format2 fmt2;
}formats ;

void io_error_check( _RIOFB_T *rfb )
{
    if ( memcmp(rfb->sysparm->_Maj_Min.major_rc,"00",2 ) ||
        memcmp ( rfb->sysparm->_Maj_Min.minor_rc,"00",2 ))
    {
        printf ( "I/O error occurred, program ends.\n" );
        exit ( 1 );
    }
}

int main(void)
{
    _RFILE *fp;
    _RIOFB_T *rfb;
    _XXIOFB_T *iofb;
    int size;
    formats buf;

    /* Open the device file. */

    if (( fp = _Ropen ( "*LIBL/T1520DDE", "ar+" )) == NULL )
    {
        printf ( "Could not open file\n" );
        exit ( 1 );
    }

```

Figure 104. T1520CDV — ILE C Source to Change the Default Device (Part 1 of 2)

```

    _Racquire ( fp,"DEVICE1" );    /* Acquire another device.      */
                                   /* Replace with the actual      */
                                   /* device name.                  */

    _Rformat ( fp,"FORMAT1" );    /* Set the record format for the */
                                   /* display file.                  */

    rfb = _Rwrite ( fp, "", 0 );  /* Set up the display.           */

    io_error_check(rfb);

    _Racquire ( fp,"DEVICE2" );    /* Acquire another device.      */

    _Rpgmdev ( fp,"DEVICE2" );    /* Change the default program    */
                                   /* device. Replace with the      */
                                   /* actual device name.          */
                                   /* Device2 implicitly acquired at */
                                   /* open.                         */

    _Rformat ( fp,"FORMAT2" );    /* Set the record format for the */
                                   /* the display file.            */

    rfb = _Rwrite ( fp, "", 0 );  /* Set up the display.           */
    io_error_check ( rfb );

    _Rreadindv ( fp, &buf, sizeof(buf), __DFT );
                                   /* Read from the first device that */
                                   /* enters data. The device becomes */
                                   /* the default program device.     */

    io_error_check ( rfb );

/* Determine which terminal responded first. */

    iofb = _Riobk ( fp );
    if ( !strcmp ( "FORMAT1 ", iofb -> rec_format, 10 ))
    {
        _Rrelease ( fp, "DEVICE1" );
    }
    else
    {
        _Rrelease(fp, "DEVICE2" );
    }
    return(0);
}

```

Figure 104. T1520CDV — ILE C Source to Change the Default Device (Part 2 of 2)

The ILE C program T1520CDV uses the `_Racquire()` function to explicitly acquire another device that is named DEVICE1. DEVICE1 becomes the current program device. The `_Rpgmdev()` function changes the current device that is named DEVICE1 to DEVICE2. The `_Rreadindv()` function reads records from DEVICE1. The `_Release()` function releases DEVICE1 and DEVICE2.

4. To run the program T1520CDV, type:

```
CALL PGM(MYLIB/T1520CDV)
```

The output is as follows:

```
Name:
Password:
```

When the application is run, a different display appears on each device. Data may be entered on both displays, but the data that is first entered is returned to the program. The output from the program is in QPRINT. For example, if the name SMITH and the address 10 MAIN ST is entered on DEVICE1 before any data is entered on DEVICE2, then the file QPRINT contains:

```
Data displayed on DEVICE1 is SMITH  10 MAIN ST
```

Note: There are two record formats that are created in the above example. One has a size of 45 characters (fmt1), and the other a size of 18 characters (fmt2). The union buf contains two record format declarations.

Using Feedback Information

You can obtain additional information about the program devices associated with your application by using OS/400 system feedback areas.

Example

The following example uses the `_Riobufk()` function.

1. To create the display file T1520DDF using the DDS source shown below, type:
`CRTDSPF FILE(MYLIB/T1520DDF) SRCFILE(QCLE/QADDSSRC) MAXDEV(2)`

```
A                                     DSPSIZ(24 80 *DS3)
A      R EXAMPLE
A      OUTPUT      5A  0  5 20
A      INPUT      20A  I  7 20
A                                     5 10'OUTPUT:'
A                                     7 10'INPUT:'
```

Figure 105. T1520DDF — DDS Source for a Feedback Display

2. To override the file STDOUT with the printer file QPRINT, type:
`OVRPRTF FILE(STDOUT) TOFILE(QPRINT)`
3. To create the program T1520FBK using the source shown below, type:
`CRTBNDC PGM(MYLIB/T1520FBK) SRCFILE(QCLE/QACSRC)`

```

/* This program illustrates how to use the _Riofbk function to access */
/* the I/O feedback area.                                          */
#include <stdio.h>
#include <recio.h>
#include <signal.h>
#include <xxfdbk.h>
#include <stdlib.h>
#include <string.h>
static void handler (int);

_RFILE *fp; /* Signal handler for _Racquire exceptions */

static void handler (int sig)
{
    _XXIOFB_T      *io_feedbk;
    _XXIOFB_DSP_ICF_T *dsp_io_feedbk;

    signal ( SIGIO, handler );

    io_feedbk = _Riofbk ( fp );
    dsp_io_feedbk = ( _XXIOFB_DSP_ICF_T *) ( (char *) (io_feedbk) +
        io_feedbk->file_dep_fb_offset );
    printf ( "Acquire failed\n" );
    printf ( "Major code: %2.2s\tMinor code: %2.2s\n",
        dsp_io_feedbk->major_ret_code, dsp_io_feedbk->minor_ret_code );
    exit ( 1 );
}

int main(void)
{
    char    buf[20];
    _RIOFB_T *rfb;

    if ( ( fp = _Ropen ( "*/LIBL/T1520DDF", "ar+") ) == NULL )
    {
        printf ( "Could not open the display file\n" );
        exit ( 2 );
    }
    signal ( SIGIO, handler );

    _Racquire ( fp, "DEVICE1" ); /* Acquire the device. DEVICE1 is */
                                /* now the default program device. */
                                /* NOTE : If the device is not */
                                /* acquired, exceptions are issued. */
    _Rformat ( fp, "EXAMPLE" ); /* Select the record format. */
    _Rwrite ( fp, "Hello", 5 ); /* Write to default program device. */

                                /* Read from default program device. */
    rfb = _Rreadn ( fp, buf, 21, __DFT );

    printf ( "user entered: %20.20s\n", buf );

    _Rclose ( fp );
    return(0);
}

```

Figure 106. T1520FBK — ILE C Source to Use Feedback Information

This program uses two typedefs `_XXIOFB_T` for common I/O feedback, and `_XXIOFB_DSP_ICF_T` for display file specific I/O feedback. A pointer to the I/O feedback is returned by `_Riofbk (fp)`.

4. To run the program `T1520FBK`, type:

```
CALL PGM(MYLIB/T1520FBK)
```

The output is as follows:

```
OUTPUT:  Hello
INPUT:
```

The `signal()` function is called before an error to establish a signal handler. If an exception occurs during the acquire operation, the signal handler is called to write the major and minor return code to `stdout`.

```
Acquire failed
Major code: 82 Minor code: AA
```

Using Subfiles

You can use subfiles to read or write a number of records to and from a display in one operation.

Example

The following subfile example uses DDS from `T1520DDG` and `T1520DDH` to display a list of names and telephone numbers.

1. To create the display file `T1520DDG` using the DDS source shown below, type:

```
CRTDSPF FILE(MYLIB/T1520DDG) SRCFILE(QCLE/QADDSSRC)
```

```
A          DSPSIZ(24 80 *DS3)
A          R SFL          SFL
A          NAME          10A B 10 25
A          PHONE          10A B  +5
A          R SFLCTL      SFLCTL(SFL)
A          SFLPAG(5)
A          SFLSIZ(26)
A          SFLDSP
A          SFLDSPCTL
A          22 25'<PAGE DOWN> FOR NEXT PAGE'
A          23 25'<PAGE UP> FOR PREVIOUS PAGE'
```

Figure 107. T1520DDG — DDS Source for a Subfile Display

2. To create the physical file T1520DDH using the DDS source shown below, type:

```
CRTPF FILE(MYLIB/T1520DDH) SRCFILE(QCLE/QADDSSRC)
```

```
      R ENTRY
      NAME          10A
      PHONE         10A
```

3. Type the following data into T1520DDH:

```
David    435-5634
Florence 343-4537
Irene    255-5235
Carrie   747-5347
Michele  634-4557
```

4. To create the program T1520SUB using the source shown below, type:

```
CRTBND CPGM(MYLIB/T1520SUB) SRCFILE(QCLE/QACSRC)
```

```
/* This program illustrates how to use subfiles. */
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>
#define LEN          10
#define NUM_RECS     20
#define SUBFILENAME  "*LIBL/T1520DDG"
#define PFILENAME    "*LIBL/T1520DDH"

typedef struct{
    char name[LEN];
    char phone[LEN];
}pf_t;
#define RECLLEN sizeof(pf_t)

void init_subfile(_RFILE *, _RFILE *);

int main(void)
{
    _RFILE      *pf;
    _RFILE      *subf;
/* Open the subfile and the physical file. */
    if ((pf = _Ropen(PFILENAME, "rr")) == NULL)
    {
        printf("can't open file %s\n", PFILENAME);
        exit(1);
    }
    if ((subf = _Ropen(SUBFILENAME, "ar+")) == NULL)
    {
        printf("can't open file %s\n", SUBFILENAME);
        exit(2);
    }
/* Initialize the subfile with records from the physical file. */
    init_subfile(pf, subf);
```

Figure 108. T1520SUB — ILE C Source to Use Subfiles (Part 1 of 2)

```

/* Write the subfile to the display by writing a record to the      */
/* subfile control format.                                         */
    _Rformat(subf, "SFLCTL");
    _Rwrite(subf, "", 0);
    _Rreadn(subf, "", 0, __DFT);

/* Close the physical file and the subfile.                         */
    _Rclose(pf);
    _Rclose(subf);
}
void init_subfile(_RFILE *pf, _RFILE *subf)
{
    _RIOFB_T    *fb;
    int         i;
    pf_t        record;

/* Select the subfile record format.                                */
    _Rformat(subf, "SFL");
    for (i = 1; i <= NUM_RECS; i++)
    {
        fb = _Rreadn(pf, &record, RECLen, __DFT);
        if (fb->num_bytes != EOF)
        {
            fb = _Rwritd(subf, &record, RECLen, i);
            if (fb->num_bytes != RECLen)
            {
                printf("error occurred during write\n");
                exit(3);
            }
        }
    }
}

```

Figure 108. T1520SUB — ILE C Source to Use Subfiles (Part 2 of 2)

This program uses `_Ropen()` to open subfile T1520DDG and physical file T1520DDH. The subfile is then initialized with records from the physical file. Subfile records are written to the display using the `_Rwritd()` function.

5. To run the program T1520SUB and see the output, type:

CALL PGM(MYLIB/T1520SUB)

```

David      435-5634
Florence   343-4537
Irene      255-5235
Carrie     747-5347
Michele    643-4557

```

```

<PAGE DOWN> FOR NEXT PAGE
<PAGE UP>  FOR PREVIOUS PAGE

```

Using Intersystem Communication Function Files

An Intersystem Communications Function (ICF) file defines the layout of the data sent and received between two programs on different systems and links you to the configuration objects that are used to communicate with a remote system. The *ICF Programming* manual contains information about ICF files.

I/O Considerations for Intersystem Communication Function Files

- An ILE C/C++ program can process ICF files as program described files or as externally described files (the system file QSYS/QICDMF contains a system-supplied record format).
- The concept of clearing or opening a file using append mode does not apply to ICF files. If you open an ICF file using append mode (ar+ or ab+), the file is opened for input and output.
- If you want to write a variable length of data, you must use the keyword VARLEN in the DDS.
- ICF locale mode can be disabled at the application level by setting the maximum program devices number to 2 or greater for all ICF files on the CRTICFF command.

Opening ICF Files as Binary Stream Files

To open an iSeries ICF file as a binary stream file for record-at-a-time processing, use the `fopen()` function with one of the following modes:

- rb
- wb and ab
- ab+

Note: The only way to create an ICF file is to use the CRTICFF command. If you use the `fopen()` function and the ICF file does not exist, a physical database file is created.

The valid keyword parameters are:

- type
- indicators

I/O Considerations for Binary Stream ICF Files

The `fwrite()` function returns the number of elements that are successfully written. When you use PDATA, the value returned by the `fwrite()` function does not take PDATA into consideration. When using PDATA, `errno` is set to ETRUNC even though all the data was successfully written.

Program Devices for Binary Stream ICF Files

The program device that is associated with ICF files is a communications session. You establish the default device by implicitly acquiring it using the `fopen()` function.

Binary Stream Functions for ICF Files

Use the following binary stream functions to process ICF files:

- `fclose()`
- `fread()`
- `fwrite()`
- `fopen()`
- `freopen()`

Opening ICF Files as Record Files

To open an iSeries ICF file as a record file, use the `_Ropen()` function with one of the following modes:

- `rr`
- `rr+` and `wr+`
- `ar+`
- `wr` and `ar`

The valid keyword parameters are:

- `indicators`
- `riofb`
- `secure`

I/O Considerations for Record ICF Files

The `_Rwrite()` function returns the number of characters that are successfully transferred across a communication line. When you use `PDATA`, unlike the `_fwrite()` function, the value that is returned by the `_Rwrite()` function (`num_bytes`) includes `PDATA`.

Program Devices for Record ICF Files

The program device that is associated with ICF files is a communications session. You establish the default device by implicitly acquiring it using the `_Ropen()` function. The implicitly acquired program device is determined by the `ACQPGMDEV` parameter on the `CRTICFF`, `OVRICFF`, or `CHGICFF` commands. If the program device name is specified on the `ACQPGMDEV` parameter the program device must be defined to the device file before it is opened. This is done by specifying the name on the `PGMDEV` parameter of the `ADDICFDEVE` or `OVRICFDEVE` commands.

If `*NONE` is specified for the `ACQPGMDEV` parameter of the `CRTICFF`, `OVRICFF`, or `CHGICFF` commands, you must explicitly acquire the program device using the `_Racquire()` function.

You can change the default program device in the following ways:

- Use the `_Racquire()` function to explicitly acquire another program device. The device that is just acquired becomes the current program device.
- Use the `_Rpgmdev()` function to change the current program device associated with a file to a previously-acquired device. This program device can be used for subsequent input and output operations to the file.
- The actual program device read becomes the default device if you read from an invited device using the `_Rreadindv()` function.
- Use the `_Rrelease()` function to release a device from the file. When you release the device, it is no longer available for I/O operations.

To release a program device, use the `_Rrelease()` function (the program device must have been previously acquired). This detaches the device from an open file; I/O operations can no longer be performed for this device. If you wish to use the device after releasing it, it must be acquired again.

All program devices are implicitly released when you close the file. If the device file has a shared open data path, the last close operation releases the program device.

Record Functions for ICF Files

Use the following record functions to process ICF files:

- `_Racquire()`
- `_Rclose()`
- `_Rdevatr()`
- `_Rfeod()`
- `_Rformat()`
- `_Rindara()`
- `_Riofbk()`
- `_Ropen()`
- `_Ropnfbk()`
- `_Rpgmdev()`
- `_Rreadindv()`
- `_Rreadn()`
- `_Rrelease()`
- `_Rupfb()`
- `_Rwrite()`
- `_Rwriterd()`
- `_Rwrread()`

Example

The following example gets a user ID and password from a source program and sends it to a target program. The target program checks the user ID and password for errors and sends a response to the source program.

Note: To run this example the target program T1520TGT must exist on a remote system. A communications line between the source system with program T1520ICF and the target system with program T1520TGT must be active. You also need Advanced Program to Program Communications (APPC).

1. To create the physical file T1520DDA, type:

```
CRTPF FILE(MYLIB/T1520DDA) SRCFILE(QCLE/QADDSSRC)
```

```
A                                UNIQUE
A      R PASSWRDF
A      USERID      8A
A      PASSWRD      10A
A      K USERID
```

Figure 109. T1520DDA — DDS Source for Password and User ID

2. To create the ICF file T1520DDB using the DDS source shown below;, type:

```
CR TICFF FILE(MYLIB/T1520DDB) SRCFILE(QCLE/QADDSSRC)
ACQPGMDEV(CAPPC2)
```

```
A      R SNDPASS
A      FLD1      18A
A      R CHKPASS
A      FLD1      1A
A      R EVOKPGM
A                                EVOKE(MYLIB/T1520TGT)
A                                SECURITY(2 'PASSWRD' +
A                                3 'USRID')
```

Figure 110. T1520DDB — DDS Source to Send Password and User ID

3. To create the ICF file T1520DDC using the DDS source shown below, type:

```
CR TICFF FILE(MYLIB/T1520DDC) SRCFILE(QCLE/QADDSSRC) ACQPGMDEV(CAPPC1)
```

A	R RCVPASS	
A	UID	8A
A	PWD	10A
A	R VRYPASS	
A	CHKPASS	1A

Figure 111. T1520DDC — DDS Source to Receive Password and Userid

4. Create an intrasystem device INTRAC. Type:
CRTDEVINTR DEVD(INTRAC) RMTLOCNAME(INTRAC) ONLINE(*NO)
5. Vary on the intrasystem device INTRAC. Type:
VRYCFG CFGOBJ(INTRAC) CFGTYPE(*DEV) STATUS(*ON) RANGE(*OBJ)
6. To add a program device entry for ICF file T1520DDB, type:
ADDICFDEVE FILE(MYLIB/T1520DDB) PGMDEV(CAPPC2) RMTLOCNAME(CAPPC1)
MODE(CAPPCMOD)
7. To add a program device entry for ICF file T1520DDC, type:
ADDICFDEVE FILE(MYLIB/T1520DDC) PGMDEV(CAPPC1) RMTLOCNAME(*REQUESTER)
MODE(CAPPCMOD)
8. To create the program T1520ICF using the source shown below, type:
CRTBNDC PGM(MYLIB/T1520ICF) SRCFILE(QCLE/QACSRC)

```

/* This program sends a userid and password to a target program      */
/* on another system. The target program returns the userid and     */
/* password. This program verifies the returned values.             */
#include <stdio.h>
#include <recio.h>
#include <stdlib.h>
#include <string.h>
#define ID_SIZE      8
#define PASSWD_SIZE 10
#define RCD_SIZE     ID_SIZE + PASSWD_SIZE
#define ERROR        '2'
#define VALID        '1'

_RFILE *fp;
void ioCheck(char *majorRc)
{
    if ( memcmp(majorRc, "00", 2) != 0 )
    {
        printf("Fatal I/O error occurred, program ends\n");
        _Rclose(fp);
        exit(1);
    }
}

int main(void)
{
    _RIOFB_T *fb;
    char      idPass[RCD_SIZE];
    char      buf[RCD_SIZE + 1];
    char      passwordCheck=ERROR;

```

Figure 112. T1520ICF — ILE C Source to Send and Receive Data (Part 1 of 2)

```

/* Open the source file T1520DDB.                                     */
if ( (fp = _Ropen("LIBL/T1520DDB", "ar+")) == NULL )
{
    printf("Could not open SOURCE ICF file\n");
    exit(2);
}

/* Start the target program T1520TGT.                                */
_Racquire(fp, "DEV1");                                              /* acquire device */
_Rformat(fp, "EVOKPGM");
fb = _Rwrite(fp, "", 0);
ioCheck(fb->sysparm->_Maj_Min.major_rc);

/* Get the user-id and password.                                     */
memset(idPass, ' ', RCD_SIZE);
printf("Enter user-id (maximum 8 characters):\n");
scanf("%s", buf);
memcpy(idPass, buf, strlen(buf));
printf("Enter password (maximum 10 characters):\n");
scanf("%s", buf);
memcpy(idPass + ID_SIZE, buf, strlen(buf));

/* Send data to the TARGET program T1520TGT.                        */
_Rformat(fp, "SNDPASS");
fb = _Rwrite(fp, idPass, RCD_SIZE);
ioCheck(fb->sysparm->_Maj_Min.major_rc);

/* Receive data from TARGET program T1520TGT.                      */
_Rformat(fp, "CHKPASS");
fb = _Rreadn(fp, &passwordCheck, 1, __DFT);
ioCheck(fb->sysparm->_Maj_Min.major_rc);

/* If a problem, such as a communications line is down, occurs in the */
/* TARGET program, then end the program.                             */
/* Otherwise, print the password verification.                       */

if ( passwordCheck == ERROR )
{
    _Rclose(fp);
    exit(3);
}
else if ( passwordCheck == VALID )
{
    printf("Password valid\n");
}
else
{
    printf("Password invalid\n");
}
_Rclose(fp);
return(0);
}

```

Figure 112. T1520ICF — ILE C Source to Send and Receive Data (Part 2 of 2)

The `_Ropen()` function opens the record file T1520DDB. The `_Rformat()` function accesses the record format EVOKPGM in the file T1520DDB. The EVOKE statement in T1520DDB calls the target program T1520TGT. The `_Rformat()` function accesses the record format SNDPASS in the file T1520DDB. The user ID and password is sent to the target program

T1520TGT. The `_Rformat()` function accesses the record format CHKPASS in the file T1520DDDB. The received password and user ID is then verified.

9. To create the program T1520TGT using the following source, type:
`CRTBNDC PGM(MYLIB/T1520TGT) SRCFILE(QCLE/QACSRC)`

```

/* This program checks the userid and password.                                     */

#include <stdio.h>
#include <recio.h>
#include <string.h>
#include <stdlib.h>

#define ID_SIZE      8
#define PASSWD_SIZE 10
#define RCD_SIZE     ID_SIZE + PASSWD_SIZE
#define ERROR        '2'
#define VALID        '1'
#define INVALID      '0'

int main(void)
{
    _RFILE      *icff;
    _RFILE      *pswd;
    _RIOFB_T     *fb;

    char        rcv[RCD_SIZE];
    char        pwr[RCD_SIZE];
    char        vry;

    /* Open the TARGET file T1520DDC.                                             */

    if ( (icff = _Ropen("QGPL/T1520DDC", "ar+")) == NULL )
    {
        printf("Could not open TARGET icf file T1520DDC\n");
        exit(1);
    }

    /* Open the PASSWORD file T1520DDA.                                          */

    if ( (pswd = _Ropen("QGPL/T1520DDA", "rr")) == NULL )
    {
        printf("Could not open PASSWORD file T1520DDA\n");
        exit(2);
    }

    /* Read the information from the SOURCE program T1520ICF.                    */

    _Racquire(icff, "DEV1");
    _Rformat(icff, "RCVPASS");
    fb = _Rreadn(icff, &rcv, RCD_SIZE, __DFT);

```

Figure 113. T1520TGT — ILE C Source to Check Data is Sent and Returned (Part 1 of 2)

```

/* Check for errors and send response to SOURCE program.          */
                                                                    */

    if ( memcmp(fb->sysparm->_Maj_Min.major_rc, "00", 2) != 0 )
    {
        vry = ERROR;
    }
    else
    {
        fb = _Rreadk(pswd, &pwr, RCD_SIZE, __DFT, &rcv, ID_SIZE);

        if ( fb->num_bytes == RCD_SIZE &&
            memcmp(pwr + ID_SIZE, rcv + ID_SIZE, PASSWD_SIZE) == 0 )
        {
            vry = VALID;
        }
        else
        {
            vry = INVALID;
        }
    }
    _Rformat(icff, "VRYPASS");
    _Rwrite(icff, &vry, 1);
    _Rclose(icff);
    _Rclose(pswd);
    return(0);
}

```

Figure 113. T1520TGT — ILE C Source to Check Data is Sent and Returned (Part 2 of 2)

The `_Ropen()` function opens the file T1520DDC. The `_Ropen()` function opens the password file T1520DDA. The `_Rformat()` function accesses the record format RCVPASS in the file T1520DDC. The `_Rreadn()` function reads the password and user ID from the source program T1520ICF. Errors are checked, and a response is sent to the source program T1520ICF.

10. To run the program T1520ICF, type:

```
CALL PGM(MYLIB/T1520ICF)
```

The output is as follows:

```

Password valid
Press ENTER to end terminal session.

```

After calling the program, you may enter a user ID and password. If the password is correct, "Password valid" appears on the display; if it is incorrect, "Password invalid" appears.

Using Printer Files

A printer device file can be accessed with a program-described file (specify `SRCFILE(*NONE)` on the `CRTPRTF` command) or with an externally described file. The object type is `*FILE`. The *ADTS/400: Advanced Printer Function* manual contains information on printer files.

Program-described files allow first character forms control (FCFC). To use this, include the first character forms control code in the first position of each data record in the printer file. You must use a printer stream file and the `fwrite()` function.

To work with externally described printer files, use one of:

- DDS through the SEU or CODE/400 editor.
- Report Layout Utility(RLU) or DSU WYSIWYG tools.

I/O Considerations for Printer Files

If you wish to use First Character Forms Control, you must use program described printer files.

Opening Printer Files as Binary Stream Files

To open an iSeries printer file as a binary stream file for record-at-a-time processing, use the `fopen()` function with one of the following modes:

- `wb`
- `ab`

Note: The only way to create a printer file is to use the CRTPRTF command. If you use the `fopen()` function and the printer file does not exist, a physical database file is created.

The valid keyword parameters are:

- `type`
- `lrecl`
- `indicators`
- `recfm`

Binary Stream Functions for Printer Files

Use the following binary stream functions to process printer files:

- `fclose()`
- `fopen()`
- `freopen()`
- `fwrite()`

Opening Printer Files as Record Files

To open an iSeries printer file as a record file, use the `_Ropen()` function with one of the following modes:

- `wr`
- `ar`

The valid keyword parameters are:

- `lrecl`
- `indicators`
- `riofb`
- `secure`

Record Functions for Printer Files

Use the following record functions to process printer files:

- `_Rclose()`
- `_Rindara()`
- `_Ropnfbk()`
- `_Rfeod()`
- `_Riofbk()`
- `_Rupfb()`
- `_Rformat()`
- `_Ropen()`
- `_Rwrite()`

Example

The following example uses first character forms control in a program described printer file. Employees' names and serial numbers are read from a physical file and written to the printer file.

1. To create the printer file T1520FCP, type:

```
CRTPRTF FILE(MYLIB/T1520FCP) CTLCHAR(*FCFC)
CHLVAL((1(13)))
```
2. To create the physical file T1520FCI, type:

```
CRTPF FILE(MYLIB/T1520FCI) RCDLEN(30)
```
3. Type the names and serial numbers as follows into T1520FCI:

```
Jim Roberts      1234567890
Karen Smith      2314563567
John Doe         5646357324
```
4. To create the program T1520FCF using the source shown below, type:

```
CRTBNDC PGM(MYLIB/T1520FCF) SRCFILE(QCLE/QACSRC)
```

```
/* This program illustrates how to use a printer stream file, the */
/* _fwrite() function and the first character forms control.      */
#include <stdio.h>
#include <string.h>
#define BUF_SIZE 53
#define BUF_OFFSET 20

int main(void)
{
    FILE      *dbf;
    FILE      *prtf;
    char buf   [BUF_SIZE];
    char tmpbuf [BUF_SIZE];

    /* Open the printer file using the first character forms control. */
    /* recfm and lrecl are required.                                   */
    prtf = fopen ("*LIBL/T1520FCP", "wb type=record recfm=fa lrecl=53" );
    dbf  = fopen ("*LIBL/T1520FCI", "rb type=record blksize=0" );

    /* Print out the header information.                               */
    memset ( buf, ' ', BUF_SIZE );

    /* Use channel value 1.                                           */
    strncpy ( buf, "1 EMPLOYEE INFORMATION",47 );
    fwrite ( buf, 1, BUF_SIZE, prtf );

    /* Use single spacing.                                           */
    strncpy ( buf, "-----",47 );
    fwrite ( buf, 1, BUF_SIZE, prtf );
    /* Use triple spacing.                                           */
    strncpy ( buf, "- NAME SERIAL NUMBER"
                ,BUF_SIZE );
    fwrite ( buf, 1, BUF_SIZE, prtf );
    strncpy ( buf, "----"
                ,BUF_SIZE );
    fwrite ( buf, 1, BUF_SIZE, prtf );
```

Figure 114. T1520FCF — ILE C Source to Use First Character Forms Control (Part 1 of 2)


```

/* Print out the employee information.                                */
while ( fread ( tmpbuf, 1, BUF_SIZE, dbf ))
{
    memset ( buf, ' ', BUF_SIZE );

/* Use double spacing.                                             */
    buf[0] = '0';
    strncpy ( buf + BUF_OFFSET, tmpbuf, strlen(tmpbuf) );
    fwrite ( buf, 1, BUF_SIZE, prtfd );
}
fclose ( prtfd );
fclose ( dbf );
}

```

Figure 114. T1520FCF — ILE C Source to Use First Character Forms Control (Part 2 of 2)

The `fopen()` function opens the printer stream file T1520FCP using record at a time processing. The `fopen()` function also opens the physical file T1520FCI for record at a time processing. The `strncpy()` function copies the records into the print buffer. The `fwrite()` function prints out the employee records.

- To run the program T1520FCF, type:
CALL PGM(MYLIB/T1520FCF)

The output file is as follows:

EMPLOYEE INFORMATION	
NAME	SERIAL NUMBER
-----	-----
Jim Roberts	1234567890
Karen Smith	2314563567
John Doe	5646357324

The printed output file is as follows:

EMPLOYEE INFORMATION	
NAME	SERIAL NUMBER
-----	-----
Jim Roberts	1234567890
Karen Smith	2314563567
John Doe	5646357324

Writing to a Tape File

You can write records to a tape file. A tape file is a device file that is used for tape units. The object type is `*FILE`. The *Tape and Diskette Device Programming* manual contains information on tape files.

I/O Considerations for Tape Files

An ILE C/C++ program can only process tape files sequentially. An ILE C/C++ program can only process tape files as program described files.

Opening Tape Files as Binary Stream Files

To open an iSeries tape file as a binary stream file for record-at-a-time processing, use the `fopen()` function with one of the following modes:

- `rb`
- `wb`
- `ab`

Note: The only way to create a tape file is to use the `CRTTAPF` command. If you use the `fopen()` function and the tape file does not exist, a physical database file is created.

The valid keyword parameters are:

- `type`
- `lrecl`
- `recfm`
- `blksize`

I/O Considerations for Binary Stream Tape Files

Blocking Binary Stream Tape Files: If your program processes tape files, performance can be improved if records are blocked.

Note: The value you specify on the `blksize` parameter for the `fopen()` function overrides the one you specified on the `CRTTAPF` or `CHGTAPF` commands. You can still override the `BLKLEN` parameter with the `OVRTAPF` command.

If you specify 0 on either `BLKLEN` or `blksize` the system calculates a block size for you. You can specify a value on either parameter of between 0 and 32 767 characters.

Binary Stream Functions for Tape Files

Use the following binary stream functions to process tape files:

- `fclose()`
- `fread()`
- `fwrite()`
- `fopen()`
- `freopen()`

Opening Tape Files as Record Files

To open an iSeries tape file as a record file, use the `_Ropen()` function with one of the following modes:

- `rr`
- `wr`
- `ar`

The valid keyword parameters are:

- `blkrcd`
- `lrecl`
- `secure`
- `riofb`

I/O Considerations for Record Tape Files

Using `_Rfeod()`: The `_Rfeod()` function is valid for files opened for input and output operations with tape record files. For input operations, it returns end-of-file and positions the tape at the last volume in the file. For output operations, it forces all unbuffered data to be written to the tape.

Using _feov: The _Rfeov() function is valid for tape record files opened for input and output operations. For input operations, it signals the end-of-file and positions the tape at the next volume. For output operations, any unwritten data is forced to the tape. An end-of-volume trailer is written to the tape which means that no data can be written after this trailer. Any write operations that take place after the _Rfeov() function occur on a new volume.

Blocking Record Tape Files: If your program processes tape files, performance can be improved if I/O operations are blocked. To block records, use the blkrcd=Y keyword on the _Ropen() function.

Record Functions for Tape Files

Use the following record functions to process tape files:

- _Rclose()
- _Rfeod()
- _Rfeov()
- _Riofbk()
- _Ropen()
- _Ropfbk()
- _Rreadn()
- _Rupfb()
- _Rwrite()

Example

The following example illustrates how to write to a tape file.

1. To create the tape file T1520TPF, type:

```
CRTTAPF FILE(MYLIB/T1520TPF) DEV(TAP01) SEQNBR(*END)
LABEL(CSOURCE) FILETYPE(*SRC)
```
2. To create the source physical file QCSRC with the member CSOURCE, type:

```
CRTSRCPF FILE(MYLIB/QCSRC) MBR(CSOURCE)
```

The CRTSRCPF command creates the physical file QCSRC with member CSOURCE in MYLIB. The following statements are copied to the tape file:

```
/* This program SQITF is called by the command SQUARE. This      */
/* program then calls another ILE C program SQ to perform      */
/* calculations and return a value.                             */
#include <stdio.h>
#include <decimal.h>
#pragma linkage(SQ, OS)      /* Tell compiler this is external call, */
                             /* do not pass by value.          */
int SQ(int);
int main(int argc, char *argv[])
{
    int *x;
    int result;
    x = (int *) argv[1];
    result = SQ(*x);
    /* Note that although the argument is passed by value, the compiler */
    /* copies the argument to a temporary variable, and the pointer to */
    /* the temporary variable is passed to the called program SQ.      */
    printf("The SQUARE of %d is %d\n", *x, result);
}
```

Figure 115. Sample Source Statements for Program T1520TAP

3. To create the program T1520TAP using the source shown below, type:

```
CRTBNDC PGM(MYLIB/T1520TAP) SRCFILE(QCLE/QACSRC)
```

```

/* This program illustrates how to write to a tape file.          */
                                                                    */

#include <stdio.h>
#include <string.h>
#include <recio.h>
#include <stdlib.h>
#define RECLLEN 80

int main(void)
{
    _RFILE *tape;
    _RFILE *fp;
    char    buf [92];
    int     i;

/* Open the source physical file containing the C source.          */
                                                                    */

```

Figure 116. T1520TAP — ILE C Source to Write to a Tape File (Part 1 of 2)

```

    if (( fp = _Ropen ( "*LIBL/QCSRC(CSOURCE)", "rr blkrcd=y" )) == NULL )
    {
        printf ( "could not open C source file\n" );
        exit ( 1 );
    }

/* Open the tape file to receive the C source statements          */
                                                                    */

    if (( tape = _Ropen ( "*LIBL/T1520TPF", "wr lrecl=92 blkrcd=y" )) == NULL )
    {
        printf ( "could not open tape file\n" );
        exit ( 2 );
    }

/* Read the C source statements, find their sizes                  */
                                                                    */
/* and add them to the tape file.                                  */
                                                                    */

    while (( _Readn ( fp, buf, sizeof(buf), __DFT )) -> num_bytes != EOF )
    {
        memmove ( buf, buf+12, RECLLEN );
        _Rwrite ( tape, buf, RECLLEN );
    }

    _Rclose ( fp );
    _Rclose ( tape );
    return(0);
}

```

Figure 116. T1520TAP — ILE C Source to Write to a Tape File (Part 2 of 2)

This program opens the source physical file T1520TPF. The `_Ropen()` function file QCSRC contains the member CSOURCE with the source statements. The `_Ropen()` function opens the tape file T1520TPF to receive the C source statements. The `_Readn()` function reads the C source statements, finds their sizes, and adds them to the tape file T1520TPF.

4. To run the program T1520TAP, type:
CALL PGM(MYLIB/T1520TAP)

After you run the program, the tape file contains the source statements from CSOURCE.

Writing to a Diskette File

You can write records to a diskette file.

A diskette file is a device file that is used for a diskette unit. The object type is *FILE. The *Tape and Diskette Device Programming* manual contains information on diskette files.

I/O Considerations for Diskette Files

A diskette unit can only be accessed with a program described file. An ILE C/C++ program can only process a diskette file sequentially.

The concept of clearing a file or opening a file using append mode does not apply to diskette files.

The diskette file label name is required when the file is opened. You specify this label name using the Override Diskette File (OVRDKTF) command.

If the diskette file is opened for input and:

- if the `lrecl` parameter is not specified or is specified as zero, the record length in the data file label on the name on the diskette is used to determine the length of the records to read.
- if the `lrecl` parameter is greater than the length of the records on the diskette file, the records that are read are padded with blanks.
- if the `lrecl` parameter is less than the length of the records on the diskette file, the records that are read are truncated.
- if the file type in the diskette file is a source file, a date and sequence number (12 bytes) is appended at the beginning of each record. You must remove these when writing the record and add 12 bytes to the `lrecl` parameter on the open statement.

Note: Output may not always result in an I/O operation to a diskette file. The I/O buffer must contain enough data to fill an entire track on a diskette.

When opening a diskette file for output, any files existing on the diskette are deleted if the data file expiration date is less than or equal to the system date.

Opening Diskette Files as Binary Stream Files

To open an iSeries diskette file as a binary stream file for record-at-a-time processing, use the `fopen()` function with one of the following modes:

- `rb`
- `wb`

Note: The only way to create a diskette file is to use the CRTDKTF command. If you use `fopen()` and the diskette file does not exist, a physical database file is created.

The valid keyword parameters are:

- type
- lrecl
- blksize

I/O Considerations for Binary Stream Diskette Files

Blocking Binary Stream Diskette Files: If your program processes diskette files, performance can be improved if I/O operations are blocked. If you do not specify a value for the blksize parameter or if you specify blksize=0 on fopen(), the system calculates a number of records to be transferred as a block to your program.

Binary Stream Functions for Diskette Files

Use the following binary stream functions to process diskette files:

- fclose()
- fread()
- fwrite()
- fopen()
- freopen()

Opening Diskette Files as Record Files

To open an iSeries diskette file as a record file, use the _Ropen() function with one of the following modes:

- rr
- wr

The valid keyword parameters are:

- blkrcd
- lrecl
- secure
- riofb

I/O Considerations for Record Diskette Files

The _Rfeod() function is valid for diskette record files opened for input and output operations. It signals the end-of-file. For output operations, it does not write any data.

Read and Write Record Diskette Files: If you read from a diskette file, the next sequential record in the diskette file is processed. Use the _Rreadn() function for reading diskette files and the _Rwrite() function for writing to diskette files.

Blocking Record Diskette Files: If your program processes diskette files, performance can be improved if records are blocked. If you specify blkrcd=Y on _Ropen(), the system calculates a number of records to be transferred as a block to your program.

Record Functions for Diskette Files

Use the following record functions to process diskette files:

- _Rclose()
- _Ropen()
- _Rupfb()
- _Rfeod()
- _Ropfbk()
- _Rwrite()
- _Riofbk()
- _Rreadn()

Example

The following example shows how to write to a diskette file.

1. To create the diskette file T1520DKF, type:

```
CRTDKTF FILE(MYLIB/T1520DKF) DEV(DKT02) LABEL(FILE1)
EXCHTYPE(*I) SPOOL(*NO)
```

2. Type:

```
CRTPF FILE(MYLIB/T1520DDI) SRCFILE(QCLE/QADDSSRC)
SHARE(*YES)
```

To create the physical file T1520DDI using the following DDS source:

A	R CUST	
A	NAME	20A
A	AGE	3B
A	DENTAL	6B

3. Type the following records into the database file T1520DDI:

Dave Bolt	35 350
Mary Smith	54 444
Mike Tomas	25 545
Alex Michaels	32 512

4. To select only records that have a value greater than 400 in the DENTAL field, type:

```
OPNQRYF FILE((MYLIB/T1520DDI)) QRYSLT('DENTAL *GT 400') OPNSCOPE(*JOB)
```

5. To create the program T1520DSK using the source shown below, type:

```
CRTBNDC PGM(MYLIB/T1520DSK) SRCFILE(QCLE/QACSRC)
```

```
/* This program illustrates how to write to a diskette file.          */
/*
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

#define BUF_SIZE 30

int main(void)
{
    _RFILE *dktf;
    _RFILE *dbf;
    char buf [BUF_SIZE];

    /* Open the diskette file                                          */
    if (( dktf = _Ropen ( "*LIBL/T1520DKF", "wr blkrcd=y lrec1=100" )) == NULL )
    {
        printf ( "DISKETTE file did not open \n" );
        exit ( 1 );
    }
}
```

Figure 117. T1520DSK — ILE C Source to Write Records to a Diskette File (Part 1 of 2)

```

/* Open the database file.                                     */
                                                                   
    if ( ( dbf = _Ropen ( "LIBL/T1520DDI", "rr" ) ) == NULL )
    {
        printf ( "DATABASE file did not open\n" );
        exit ( 2 );
    }

/* Copy all the database records meeting the OPNQRYF selection */
/* criteria to the diskette file.                               */

    while ( ( _Readn ( dbf, buf, BUF_SIZE, __DFT ) ) -> num_bytes != EOF )
    {
        _Rwrite ( dktf, buf, BUF_SIZE );
    }
    _Rclose ( dktf );
    _Rclose ( dbf );
}

```

Figure 117. T1520DSK — ILE C Source to Write Records to a Diskette File (Part 2 of 2)

The `_Ropen()` function opens the diskette file T1520DKF and the database file T1520DDI. The `_Readn()` function reads all database records. The `_Rwrite()` function copies all database records that have a value > 400 in the dental field to the diskette file T1520DKF.

6. To run the program T1520DSK, type:
CALL PGM(MYLIB/T1520DSK)

The output to the diskette file is as follows:

Mary Smith	444
Mike Tomas	545
Alex Michaels	512

After you run the program, the diskette file contains only the records that satisfied the selection criteria.

Using Save Files

A **save file** is a file allocated in auxiliary storage that can be used to store saved data on disk (without requiring diskettes or tapes), or to receive objects sent through the network. The object type is *FILE. The *Backup and Recovery* manual contains information on save files.

I/O Considerations for Save Files

An ILE C/C++ program can only process save files sequentially. All records that are read or are written must be 528 characters in length. Any records that are written to another save file cannot be changed by the ILE C/C++ program.

Opening Save Files as Binary Stream Files

To open an iSeries save file as a binary stream file for record-at-a-time processing, use the `fopen()` function with one of the following modes:

- rb
- wb
- rb

Note: The only way to create a save file is to use the CRTSAVF command. If you use the `fopen()` function with a mode of `wb` or `ab` and the save file does not exist, a physical database file is created.

The valid keyword parameters are:

- `lrecl`
- `type`

I/O Considerations for Binary Stream Save Files

There are no special considerations for binary stream save files.

Binary Stream Functions for Save Files

Use the following binary stream functions to process save files:

- `fclose()`
- `fread()`
- `fwrite()`
- `fopen()`
- `freopen()`

Opening Save Files as Record Files

To open an iSeries save file as a record file, use the `_Ropen()` function with one of the following modes:

- `rr`
- `wr`
- `ar`

The valid keyword parameters are:

- `lrecl`
- `riofb`
- `secure`

I/O Considerations for Record Save Files

If a save file is opened for input, the `_Rfeod()` function returns an end-of-file to your program. If a save file is opened for output, the `_Rfeod()` function ensures that any data that is written to the file is forced to auxiliary storage. If you want to continue reading from or writing to the save file after calling this function, you must close the file and open it again.

Record Functions for Save Files

The following record functions can be used to process save files:

- `_Rclose()`
- `_Ropen()`
- `_Rupfb()`
- `_Rfeod()`
- `_Ropnfbk()`
- `_Rwrite()`
- `_Riofbk()`
- `_Rreadn()`

Part 6. Working with iSeries Features

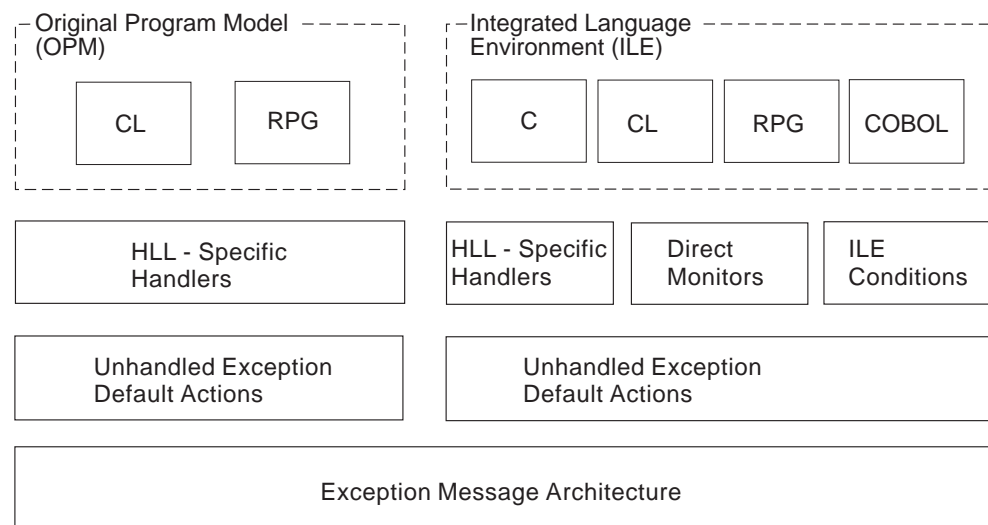
This part describes how to:

- Understand the ILE control boundary and exception handling
- Use try-catch-throw, ILE condition handlers, C signal function and cancel handlers
- Percolate and promote exceptions
- Monitor for iSeries system exceptions
- Declare and use iSeries pointers
- Use packed decimal data types in your C programs
- Code program and procedure calls to other AS/400 programming languages
- Pass parameters to other iSeries programming languages
- Qualify library calls
- Use teraspace storage

Chapter 13. Handling Exceptions in Your Program

This chapter describes how to:

- Handle exceptions
 - Check the return value of a function
 - Check the errno value
 - Check the system exceptions for stream files
 - Check the system exceptions for record files
 - Try-catch-throw for C++
 - Posix signals
- Use direct monitor handlers
- Use Integrated Language Environment condition handlers
- Use high level language (HLL)-specific handlers. For C, the HLL handler mechanism is `signal`



RV3W101-0

Figure 118. Error Handling for OPM and ILE

For OPM programs, language specific error handling provides one or more handling routines for each call stack entry. The system calls the appropriate routine when an exception is sent to an OPM program.

Language-specific error handling in ILE C/C++ provides the same capabilities. ILE C/C++, however, has additional types of exception handlers. These types of handlers allow you to change the exception message to indicate that the exception is handled and to bypass the language-specific error handling. The additional types of handlers for ILE C/C++ are:

- Direct monitor handler
- Integrated Language Environment condition handler

The **call message queue** facilitates the sending and receiving of informational messages and exception messages between calls on the call stack. A call appears on

the job call stack when OPM *PGM objects are called or when ILE procedures and functions are called. ILE *PGM objects and *SRVPGM objects never appear on the call stack. Only procedures or functions within ILE *PGM or *SRVPGM objects can appear on the call stack (when they are called).

A **message queue** exists for every call stack entry within each iSeries job. As soon as a new entry appears in the call stack, the system creates a new call message queue. In ILE C/C++, the name of the procedure identifies the call message queue. If the procedure names are not unique, you can specify the module name, program name, or service program as well. If you determine that your handler does not recognize an exception message, the exception message can be percolated to the next handler.

Percolation occurs when an unhandled exception message is moved to the call message queue of the previous call message stack entry.

If the function check message is percolated to the control boundary, ILE considers the application to have ended with an unexpected error. ILE defines a generic failure exception message for all languages. This message is CEE9901 and ILE sends this message to the caller of the control boundary.

The following are the only types of messages that are considered to be **exception messages**

(*ESCAPE)	Indicates an error that caused a program to end abnormally.
(*STATUS)	Describes the status of work that the program is in the process of doing.
(*NOTIFY)	Describes a condition that requires corrective action or reply from calling program.
Function Check	Describes an ending condition that the program has not expected.

You can monitor all of these exception message types by using the #pragma exception handler directive. ¹

To process exceptions, the system uses a handle cursor and a resume cursor.

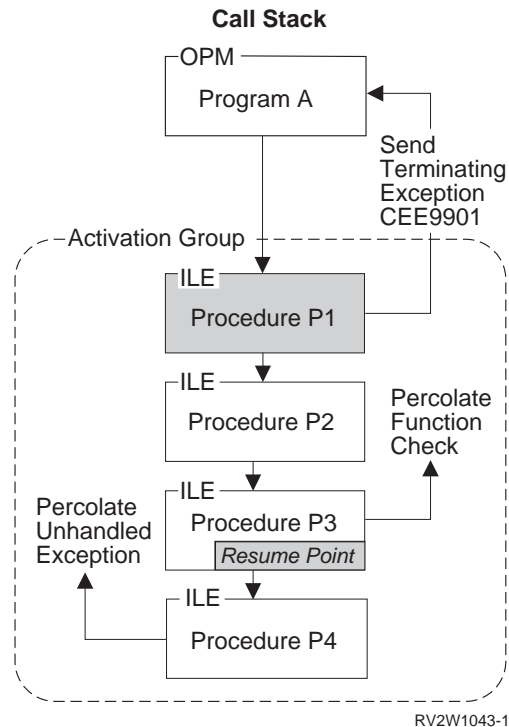
The **handle cursor** describes the location of the current exception handler. As the system searches for an available exception handler, it moves the handle cursor to the next handler in the exception handler list. The list may contain direct monitor handlers, Integrated Language Environment condition handlers, and HLL-specific handlers.

A **suspend point** within a function is a point at which control is suspended within that function. A function may be suspended due to an exception that is occurring at that point, or more commonly, due to a call to another function.

The **resume cursor** describes the point at which a program will resume after handling an exception. This is initially set to the instruction following the suspend point of the call stack entry that caused the exception.

1. See the Record Input and Output Error Macro to Exception Mapping table in the Run-Time Considerations chapter of the *ILE C for AS/400 Run-Time Library Reference*. It provides a list of exception messages generated by the ILE C/C++ record I/O functions.

The **resume point** for a suspended call stack entry of a function is the location in the function that will get control when the call stack entry resumes. The resume point is initially set to the instruction following the suspend point of the call stack entry.



Handling Exceptions

Checking the Return Value of a Function

Your program should check the function return value to verify that the function has completed successfully.

The following example illustrates how to check the return value of a function.

```

#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    FILE *fp;
    if (( fp = fopen ( "MYLIB/QCSRC(TEST)", "ab" )) == NULL )
    {
        printf ("Cannot open file QCSRC(TEST)\n");
        exit (99);
    }
}

```

Figure 120. ILE C Source to Check for the Return Value of `fopen()`

Checking the Errno Value

The `<errno.h>` header file contains various declarations for defined error conditions. Many C functions set `errno` to specific values, depending on the type of error. These values are also defined in the `<errno.h>` header file. The implementation of `errno` contains a function call. The *ILE C for AS/400 Run-Time Library Reference* contains a list of `errno` macro values.

Your program can use the `strerror()` and `perror()` functions to print the value of `errno`. The `strerror()` function returns a pointer to an error message string that is associated with `errno`. The `perror()` function prints a message to `stderr`. The `perror()` and `strerror()` functions should be used immediately after a function is called since subsequent calls might alter the `errno` value.

Note: Your program should always initialize `errno` to 0 (zero) before calling a function because `errno` is not reset by any library functions. Check for the value of `errno` immediately after calling the function that you want to check. You should also initialize `errno` to zero after an error has occurred.

Example

The following example illustrates how to check the `errno` value.

```

#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
void main(void)
{
    FILE *fp;
    errno = 0;
    fp = fopen("Nofile", "r");
    if ( errno != 0 ) {
        perror("Error occurred while opening file.\n");
        exit(1);
    }
}

```

Figure 121. ILE C Source to Check the `errno` Value for `fopen()`

Checking the Global Variable `_EXCP_MSGID`

The global variable `_EXCP_MSGID` is set whenever a stream or record I/O function gets an exception. The global variable `_EXCP_MSGID`, declared in the `<stddef.h>` header file, contains the exception message ID. See the the Record Input and Output Error Macro to Exception Mapping table in the Run-Time Considerations chapter of the *ILE C for AS/400 Run-Time Library Reference* for information about the `_EXCP_MSGID` setting after an OS/400 exception.

Checking the System Exceptions for Stream Files

In addition to checking the return value of a function or the `errno` value, you can check the major and minor return code to detect stream file errors. If your program processes display, ICF, or printer files as stream files, you can check the external variable `_C_Maj_Min_rc`, which is defined in `<stdio.h>`. The definition of this structure is:

```
typedef struct _Major_Minor_rc
{
    char major_rc[2];
    char minor_rc[2];
} _Major_Minor_rc;
extern _Major_Minor_rc _C_Maj_Min_rc;
```

Stream I/O functions trap the SIGIO signal.

Note: Signal handlers that are registered for SIGIO are not called for exceptions that are generated when processing stream files.

The following `errno` macros indicate an OS/400 system exception:

- `EIOERROR`: a nonrecoverable I/O error has occurred.
- `EIOECERR`: a recoverable I/O error has occurred.

Checking the System Exceptions for Record Files

In addition to checking the return value of a function or the `errno` value, you can check some values in the `_RIOFB_T` structure, defined in `<recio.h>`, to detect record file errors. The definition of the `_RIOFB_T` structure is shown below:

```
typedef struct {
    unsigned char *key;
    _Sys_Struct_T *sysparm;
    unsigned long rrn;
    long num_bytes;
    short blk_count;
    char blk_filled_by;
    int dup_key : 1;
    int icf_locate : 1;
    int reserved1 : 6;
    char reserved2[20];
} _RIOFB_T;
```

If your program processes display, ICF, or printer files as record files, you can check the `num_bytes` field in the `_RIOFB_T` structure and the major/minor return code fields in the `sysparm` area of the `_RIOFB_T` structure. If your program processes database files as stream files, you can check the values in some fields in the `_RIOFB_T` structure, which is defined in `<recio.h>`.

The num_bytes field and the sysparm field contain information regarding record file I/O errors.

The num_bytes field indicates if the I/O operation was successful.

The sysparm field points to a structure that contains the major and minor return code for display, ICF, or printer files. The definition of _Sys_Struct_T structure is shown below:

```
typedef struct {          /* System specific information */
    void          *sysparm_ext;
    _Maj_Min_rc_T _Maj_Min;
    char          reserved1[12];
} _Sys_Struct_T;
```

The definition of _Maj_Min_rc_T structure is:

```
typedef struct {
    char major_rc[2];
    char minor_rc[2];
} _Maj_Min_rc_T;
```

Example

The following example shows a number of ways to handle errors, including checking the major/minor return code, checking errno, getting error information from the _EXCP_MSGID global variable, and signal handling with signal.

1. To create the display file T1520DDJ using the DDS source shown below, type:
CRTDSPF FILE(MYLIB/T1520DDJ) SRCFILE(QCLE/QADDSSRC)

```
A                                     DSPSIZ(24 80 *DS3)
A                                     INDARA
A          R PHONE
A                                     CF03(03 'EXIT')
A                                     CF05(05 'REFRESH')
A                                     7 28'Name:'
A          NAME          11A B 7 34
A                                     9 25'Address:'
A          ADDRESS      20A B 9 34
A                                     11 25'Phone #:'
A          PHONE_NUM    8A B 11 34
A                                     1 35'PHONE BOOK'
A                                     DSPATR(HI)
A                                     16 19'<ENTER> : Saves changes'
A                                     17 21'f3   : Exits with changes saved'
A                                     18 21'f5   : Brings back original field values'
A                                     21 32'Screen refreshed'
A 05                                     DSPATR(HI)
A 05
```

Figure 122. T1520DDJ — DDS Source for a Phone Book Display

2. To create the program T1520EHD using the source shown below, type:
CRTBNDC PGM(MYLIB/T1520EHD) SRCFILE(QCLE/QACSRC)

Program T1520EHD uses signal () function to raise a SIGIO signal.


```

/* This program illustrates how to:                                     */
/*      - check the major/minor return code                           */
/*      - check the errno global variable                             */
/*      - get error information from the                               */
/*      _EXCP_MSGID global variable                                   */
/*      - use the signal function.                                     */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stddef.h>
#include <errno.h>
#include <signal.h>
#include <recio.h>

#define IND_ON  '1'
#define IND_OFF '0'
#define HELP    0
#define EXIT    2
#define REFRESH 4
#define FALSE   0
#define TRUE    1

typedef struct PHONE_LIST_T{
    char name[11];
    char address[20];
    char phone[8];
}PHONE_LIST_T;
void error_check(void);
/* The error checking routine.                                         */
void error_check(void)
{
    if (errno == EIOERROR || errno == EIORECERR)
        printf("id:%7.7s\n", _EXCP_MSGID);
    if (strncmp(_Maj_Min_rc.major_rc,"00",2) ||
        strncmp(_Maj_Min_rc.major_rc,"00",2))

        printf("Major : %2.2s\tMinor : %2.2s\n",
            _Maj_Min_rc.major_rc,_Maj_Min_rc.minor_rc);
    errno = 0;
}
/* The signal handler routine.                                         */

void sighd(int sig)
{
    signal(SIGIO,&sighd);
}

/* M A I N   L I N E                                                 */

int main(void)
{
    FILE *dspf;
    PHONE_LIST_T phone_inp_rec,
        phone_out_rec = { "Smith, John",
            "2711 Westsyde Rd.  ",
            "721-9729" };

```

Figure 123. T1520EHD — ILE C Source to Handle Exceptions (Part 1 of 2)

```

_SYSindara indicator_area;
int ret_code;

errno = 0;

signal(SIGIO,&signd); /* Register signd as a handler for I/O exceptions */

if ((dspf = fopen("*/LIBL/T1520DDJ", "ab+ type=record indicators=y"))
    == NULL)
{
    printf("Display file could not be opened");
    exit(1);
}
_Rindara((_RFILE *) dspf,indicator_area);
_Rformat((_RFILE *) dspf,"PHONE");

memset(indicator_area,IND_OFF,sizeof(indicator_area));
do
{
    ret_code = fwrite(&phone_out_rec,1,sizeof(phone_out_rec),dspf);
    error_check(); /* Write the records to the display file. */
    ret_code = fread(&phone_inp_rec,1,sizeof(phone_inp_rec),dspf);
    error_check(); /* Read the records from the display file. */
    if (indicator_area[EXIT] == IND_ON)
        phone_inp_rec = phone_out_rec;
}
while (indicator_area[REFRESH] == IND_ON);

_Rclose((_RFILE *)dspf);
}

```

Figure 123. T1520EHD — ILE C Source to Handle Exceptions (Part 2 of 2)

3. To run the program T1520EHD, type:

```
CALL PGM(MYLIB/T1520EHD)
```

The output is as follows:

```

                PHONE BOOK
            Name: Smith, John
        Address: 2711 Westsyde Rd.
        Phone #: 721-9729
<ENTER> : Saves changes
        f3  : Exits with changes saved
        f5  : Brings back original field values

```

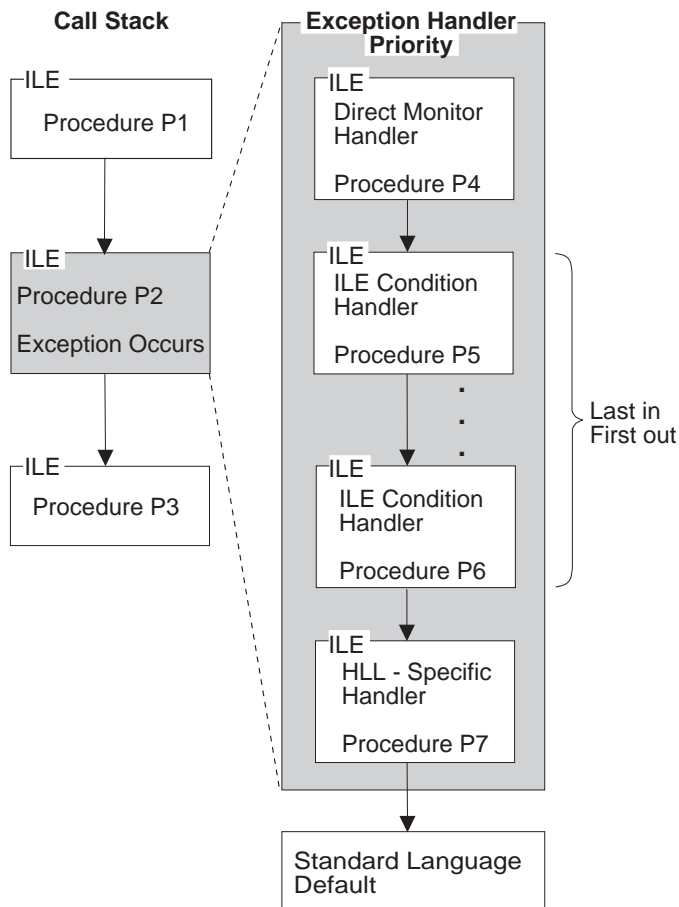
Using Exception Handlers

The types of exception handlers in ILE are:

- Posix signals.
- Direct monitor handlers, enabled with the `#pragma exception_handler` directive.
- Integrated Language Environment condition handlers that allow you to register a condition handler at run time by using the Integrated Language Environment condition handler bindable API CEEHDLR.
- HLL-specific handlers, for example, the C `signal()` function.
- Try-catch-throw for C++.

Exception handler priority becomes important if you use both language-specific error handling (C signal) and additional Integrated Language Environment exception handler types. For the call stack entry that incurred the exception, the system calls handlers in the following prioritized order:

1. Direct monitors
2. ILE condition handlers
3. `signal()`



RV2W1041-3

Figure 124. Exception Handler Priority

For portable code across multiple platforms, only the `signal()` function should be used. ILE condition handlers should be used if a consistent mechanism for handling exceptions across ILE enabled languages is required. If portability across ILE-enabled platforms is a concern, then ILE condition handlers and the `signal()` function can be used. Otherwise, all three types of handlers may be used.

Using the `signal()` function will always handle the exception implicitly (unless the signal action is `SIG_DFL`, in which case it would percolate the exception); with direct monitor handlers you either have to specify a control action that will implicitly handle the exception (`_CTLA_HANDLE`, `_CTLA_HANDLE_NO_MSG`, `_CTLA_IGNORE`, or `_CTLA_IGNORE_NO_MSG`), or you have to handle the exception explicitly within the handler function (when the control action `_CTLA_INVOKE` is specified), using either `QMHCHGEM` or an ILE condition handling API.

Note: Direct monitors are usually the fastest handlers.

The HLL-specific handler which is the signal handler in ILE C is global. It is enabled for all function calls in the activation group the `signal()` function is called in. Integrated Language Environment condition handlers and direct monitor handlers are scoped to the function that enables them or until they are disabled in that function.

The following example illustrates that if you do not want to change the state of a signal handler when the signal function returns, then you must manage the state of the signal handler explicitly.

```
#include <signal.h>
void f(void)
{
    void (*old_state)(int);
    /* Save old state of signal action */
    old_state = signal(SIGALL,handler);
    /* Other code in your application */
    /* Reset state of signal          */
    signal(SIGALL,old_state);
}
```

Figure 125. ILE C Source to Manage the State of a Signal Handler

Integrated Language Environment condition handlers and direct monitor handlers do not have this requirement because they are not global handlers.

Unhandled Exceptions

If you do not handle an exception in the call stack entry that caused the exception, it is percolated (moved) to the caller's call message queue. If the caller does not handle the exception then the message is percolated again. This continues until the exception reaches a control boundary call stack entry, at which point the system takes the default action for the unhandled exception.

If the message type is `*STATUS` the program resumes without logging the exception. If the message type is `*NOTIFY` the default reply is sent. If the message type is `*ESCAPE` then a function check is sent to the call stack entry that is pointed to by the resume cursor. If the message is a function check then the call stack is cancelled to the control boundary and CEE9901 is sent to the caller of the control boundary (the resume cursor then points to the caller of the control boundary).

Example

The following example shows an unhandled exception. An exception is sent to the `fred()` function. The `main()` function is the control boundary. The `fred()` function has no exception handlers therefore the exception is percolated to `main()`. The `main()` function has no exception handlers and `main()` is a **control boundary**, and the system takes the default action. The exception is of type `*ESCAPE` so function check is sent to the `fred()` function. The function check percolates to function `main()`, and again the default is taken. The exception is of type function check, therefore, the call stack entries of the `main()` and `fred()` functions are cancelled and the CEE9901 exception is sent to the caller of function `main()`.

```

void fred(void)
{
    char *p = NULL;
    *p = 'x';    /* *ESCAPE exception */
}
int main(void)
{
    fred();
}

```

Figure 126. ILE C Source for Unhandled Exceptions

Using Direct Monitor Handlers

Direct monitor handlers let you directly register an exception monitor around a limited number of C source statements. For ILE C, this is enabled through the `#pragma exception_handler` and `#pragma disable_handler` directives. You should include the `<except.h>` header file in your source code when using these directives. The `#pragma exception_handler` directive can only monitor the exception message types listed in the “exception messages list” on page 258.

The `#pragma exception_handler` enables a direct monitor handler from `#pragma exception_handler` to `#pragma disable_handler` without considering program logic in between. The direct monitor handler maybe either code following a label defined within the function containing the `#pragma exception_handler`, or a function.

A communications area variable may be specified on the `#pragma exception_handler`, and has a different use depending on whether the handler is a label or a function. If the handler is a label then the communications area is used as storage for the standard exception handler parameter block of type `_INTRPT_Hndlr_Parms_T` (defined in `<except.h>`).

The definition of this structure is:

```

typedef _Packed struct {
    unsigned int    Block_Size;        /* Size of the parameter block */
    _INVFLAGS_T    Tgt_Flags;         /* Target invocation flags */
    char            reserved[8];       /* reserved */
    _INVPTR        Target;            /* Current target invocation */
    _INVPTR        Source;            /* Source invocation */
    _SPCPTR        Com_Area;          /* Communications area */
    char            Compare_Data[32];  /* Compare Data */
    char            Msg_Id[7];         /* Message ID */
    char            reserved1;         /* 1 byte pad */
    _INTRPT_Mask_T Mask;              /* Interrupt class mask */
    unsigned int    Msg_Ref_Key;        /* Message reference key */
    unsigned short  Exception_Id;      /* Exception ID */
    unsigned short  Compare_Data_Len;  /* Length of Compare Data */
    char            Signal_Class;      /* Internal signal class */
    char            Priority;           /* Handler priority */
    short           Severity;          /* Message severity */
    char            reserved3[4];
    int             Msg_Data_Len;      /* Length of available message data */
    char            Mch_Dep_Data[10]; /* Machine dependent data */
    char            Tgt_Inv_Type;      /*Invocation type (in MIMCHOBS.H)*/
}

```



```

    _SUSPENDPTR    Tgt_Suspend;    /* Suspend pointer of target    */
    char           Ex_Data[48];    /* First 48 bytes of exception data */
} _INTRPT_Hndlr_Parms_T;

```

The system fills in the structure prior to giving control to the label.² If the handler is a function, the system passes a pointer to a structure of type `_INTRPT_Hndlr_Parms_T` to the function. A pointer to the communications area is available inside the structure.

The direct monitor handlers are scoped at compile time to the code between the `#pragma exception_handler` directive and the `#pragma disable_handler` directive. For example, the `#pragma exception_handler` directive is scoped to a block of code independent of the program logic.

```

volatile int ca=0;
if (ca != 0){
    #pragma exception_handler(my_handler, ca,0,_C2_MH_ESCAPE)
}
else {
    raise(SIGINT);/* Signal will be caught by my_handler */
}
#pragma disable_handler

```

Figure 127. ILE C Source to Scope Direct Monitor Handlers

The example above shows the `#pragma exception_handler` directive enabled around the call to the `raise()` function. The conditional expression `if (ca != 0)` has no effect on enabling the direct monitor handler. The logic path for the conditional expression `if (ca != 0)` is never taken, but `my_handler` is enabled.

Exception Classes

Exception classes indicate the type of exception (for example, `*ESCAPE`, `*NOTIFY`, `*STATUS`, function check) and, for machine exceptions, the low level type (for example, pointer not valid, divide by zero). Direct monitor handlers monitor for exceptions that are based on exception classes and message identifiers. The handler will get control if the exception falls into one or more of the exception classes that are specified on the `#pragma exception_handler`.

2. If the storage that is required for the exception handler parameter block exceeds the storage that is defined by `com_area` then the remaining bytes are truncated.

```

#include <except.h>
/* Just monitor for pointer not valid exceptions */
#pragma exception_handler(eh, 0, _C1_POINTER_NOT_VALID, 0)
/* Monitor for all *ESCAPE messages */
#pragma exception_handler(eh, 0, 0, _C2_MH_ESCAPE)
/* Although the following is valid, there is no need to specify */
/* _C1_POINTER_NOT_VALID because it is covered by _C2_MH_ESCAPE */
#pragma exception_handler(eh, 0, _C1_POINTER_NOT_VALID, _C2_MH_ESCAPE)
/* To monitor for only specific messages, use the extended form of */
/* pragma exception_handler. */
/* The following #pragma will only monitor for MCH3601 *ESCAPE msg. */
#pragma exception_handler (eh, 0, 0, _C2_MH_ESCAPE, _CTLA_HANDLE, "MCH3601")

```

Figure 128. ILE C Source to Use Exception Classes

All machine exceptions are mapped to the *ESCAPE type exception. To monitor for machine exceptions you can either specify the machine exception class, or specify all *ESCAPE exceptions. Macros for the iSeries machine exception classes are defined in the ILE C/C++ include file <except.h>.

You can monitor for the exception class values for class1 and class2. The value of class2 can only be one of _C2_MH_ESCAPE, _C2_MH_STATUS, _C2_MH_NOTIFY, or _C2_MH_FUNCTION_CHECK as defined in the <except.h> include file.

The Run-Time Considerations section of the *ILE C for AS/400 Run-Time Library Reference* contains a table of the exception classes.

Control Actions

The #pragma exception_handler directive allows you to specify a control action that is to be taken during exception processing. The five control actions that can be specified, as defined in the <except.h> header file, are:

_CTLA_INVOKE

This control action will cause the function that is named on the directive to be called and will not handle the exception. The exception will remain active and must be handled by using QMHCHGEM or one of the ILE condition-handling APIs.

_CTLA_HANDLE

This control action will cause the function or label that is named on the directive to get control and it will handle and log the exception implicitly. The exception will no longer be active when the handler gets control.

_CTLA_HANDLE_NO_MSG

This control action is the same as _CTLA_HANDLE except that the exception is NOT logged. The message reference key in the parameter block that is passed to the handler will be zero.

_CTLA_IGNORE

This control action will handle and log the exception implicitly and will not pass control to the handler function named on the directive; that is, the function named will be ignored. The exception will no longer be active, and processing will resume at the instruction immediately following the instruction that caused the exception.

_CTLA_IGNORE_NO_MSG

This control action is the same as _CTLA_IGNORE except that *NOTIFY messages will be logged.

The following example shows how the control action parameter can be specified on the `#pragma exception_handler` directive. The example contains code that will cause an MCH3601 (Pointer not set) exception. The control action `_CTLA_IGNORE` will cause the exception to be handled without calling the handler function. The output of this code is the message: "Passed the exception."

```
#include <except.h>
#include <stdio.h>
void myhandler(void) {
    printf("In handler - something's wrong!\n");
    return;
}
int main(void) {
    int *ip;
    volatile int com_area;
    #pragma exception_handler(myhandler, com_area, 0, _C2_ALL, \
                             _CTLA_IGNORE)

    *ip = 5;
    printf("Passed the exception.\n");
}
```

Figure 129. ILE C Source to Handle Exceptions

Specifying Message Identifiers

The `#pragma exception_handler` directive can specify one or more specific or generic message identifiers on the directive. When one or more identifiers are specified on the directive, the direct monitor handler will take effect only when an exception occurs whose identifier matches one of the identifiers on the directive.

To specify message identifiers on the directive, you have to specify a control action to be taken. The class of the exception must be in one of the classes specified on the directive. The following example shows a `#pragma exception_handler` directive that enables a monitor for a single specific message, MCH3601:

```
#pragma exception_handler (myhandler, com_area, 0, _C2_ALL, \
                           _CTLA_HANDLE, "MCH3601")
```

Following is an example of a `#pragma exception_handler` directive that enables a monitor for several floating-point exceptions:

```
#pragma exception_handler (myhandler, com_area, _C1_ALL, _C2_ALL, \
                           _CTLA_IGNORE, "MCH1206 MCH1207 MCH1209 MCH1213")
```

The ability to specify generic message identifiers can be used to simplify the directive in the previous example. In the example that follows, a monitor is enabled for any exception whose identifier begins with "MCH12":

```
#pragma exception_handler (myhandler, com_area, _C1_ALL, _C2_ALL, \
                           _CTLA_IGNORE, "MCH1200")
```

Examples

The following shows the source for a program MYPGM:

```

/* MYPGM *PGM */
#include <except.h>
#include <stdio.h>
void my_handler(_INTRPT_Hndlr_Parms_T * __ptr128 parms);
void main_handler(_INTRPT_Hndlr_Parms_T * __ptr128 parms);
void fred(void)
{
    char *p = NULL;
    #pragma exception_handler(my_handler, 0,0,_C2_MH_ESCAPE)
    *p = 'x';    /* exception */
    #pragma disable_handler
}
int main(void)
{
    #pragma exception_handler(main_handler, 0,0,_C2_MH_ESCAPE)
    fred();
}

```

Figure 130. T1520XH1 — ILE C Source to Use Direct Monitor Handlers — main()

The following example shows the source for the service program HANDLERS:

```

#include <signal.h>
#include <stdio.h>
/* HANDLERS *SRVPGM (created with activation group *CALLER) */
void my_handler(_INTRPT_Hndlr_Parms_T * __ptr128 parms)
{
    return;
}
void main_handler(_INTRPT_Hndlr_Parms_T * __ptr128 parms)
{
    printf("In main_handler\n");
}

```

Figure 131. T1520XH2 — ILE C Source to Use Direct Monitor Handlers — Service Program

In the example, the procedure main() in MYPGM registers the direct monitor handler main_handler followed by a call to fred() which registers the direct monitor handler my_handler. The fred() function gets an exception which causes my_handler to get control, followed by main_handler. The main() function is a control boundary.

The exception is considered unhandled so a function check is sent to fred(). The handlers my_handler and main_handler only handle *ESCAPE messages, so neither is called again. The function check goes unhandled at main() so the program ends abnormally and CEE9901 is sent to the caller of main().

The following example illustrates direct monitor handlers using labels instead of functions as the handlers:

```

#include <except.h>
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>
void sig_hndlr(int);
void sig_hndlr(int sig){
    printf("Signal handler should not have been called\n");
}
int main(void)
{
    int a=0;
    char *p=NULL;
    volatile _INTRPT_Hndlr_Parms_T ca;
    /* Set up signal handler for SIGFPE. The signal handler function */
    /* should never be invoked, since the exception will be handled */
    /* by the direct monitor handlers. */
    if( signal(SIGFPE,sig_hndlr) == SIG_ERR )
    {
        printf("Could not set up signal handler for SIGFPE\n");
    }
    /* The following direct monitor will */
    /* trap and handle any *ESCAPE exceptions. */
    #pragma exception_handler(LABEL_1, ca, 0, _C2_MH_ESCAPE, \
        _CTLA_HANDLE)
    /* Generate exception(divide by zero). The CTL_ACTION specified */
    /* should take effect (exception handled and logged), execution */
    /* resumes at LABEL_1. */
    a/=a;
    printf ("We should never reach this point\n");
    LABEL_1: printf("The MCH1211 exception was handled\n");
    #pragma disable_handler
    /* The following direct monitor will */
    /* only trap and handle MCH3601 exceptions */
    #pragma exception_handler(LABEL_2, ca, 0, _C2_MH_ESCAPE, \
        _CTLA_HANDLE, "MCH3601")
    /* Generate MCH3601(*ESCAPE message). The CTL_ACTION specified */
    /* should take effect (exception handled and logged), execution */
    /* resumes at LABEL_2. */
    *p='X';
    printf ("We should never reach this point\n");
    LABEL_2: printf("The MCH3601 exception was handled\n");
}

```

Figure 132. T1520XH3 — ILE C Source to Use Direct Monitors with Labels as Handlers

The following is the output:

```

The MCH1211 exception was handled
The MCH3601 exception was handled

```

The following example shows you how to use the `#pragma exception_handler` and the `signal()` function together. This example also shows how an exception is handled using SIGIO. An end-of-file message is mapped to SIGIO. The default for SIGIO is SIG_IGN. It also shows that when both a HLL-specific handler and direct monitor handler are defined, the direct monitor handler is called first.

1. To create the program T1520ICA, using the following source, type:
CRTBNDC PGM(MYLIB/T1520ICA) SRCFILE(QCLE/QACSRC)

```

/* This program illustrates how to use direct monitor handlers.      */
#include <stdio.h>
#include <signal.h>
#include <recio.h>
#include <stdlib.h>
#include <string.h>
#include <except.h> /* Include except.h even though it is included */
                    /* in the signal.h header file.                */
#define FILE_NAME  "QTEMP/MY_FILE"
#define RCD_LEN    80
#define NUM_RCD    5
#pragma datamodel(p128)
typedef struct error_code{
    int      byte_provided;
    int      byte_available;
    char      exception_id[7];
    char      reserve;
    char      exception_data[1];
}error_code_t;
static int handle_flag;
#pragma linkage(QMHCHGEM, OS)
void QMHCHGEM(_INVPTR *, int, unsigned int, char *,
              char *, int, error_code_t *);
/* The signal handler.                                              */
#pragma datamodel(pop)
static void sig_handler(int sig)
{
    printf("In signal handler\n");
    printf("Exception message ID is %7.7s\n", _EXCP_MSGID);
}
/* The direct monitor handler.                                      */
static void exp_handler(_INTRPT_Hndlr_Parms_T * __ptr128 exp_info)
{
    error_code_t      error_code;
    printf("In direct monitor handler\n");
    printf("Exception message ID is %3.3s%04x\n",
           exp_info->Compare_Data,
           (unsigned) exp_info->Exception_Id);
/* Call QMHCHGEM API to handle the exception.                      */
    if ( handle_flag )
    {
        error_code.byte_provided = 8;
        QMHCHGEM(&(exp_info->Target),; 0, exp_info->Msg_Ref_Key,
                 "HANDLE ", "", 0, &error_code);
    }
}
/* The function to read a file.                                     */
static void read_file(_RFILE *fp)
{
    int i = 1;
    while ( _Rreadn(fp, NULL, RCD_LEN, __DFT)->num_bytes != EOF )
    {
        printf("Read record %d\n", i++);
    }
}

```

Figure 133. T1520ICA — ILE C Source to Use Direct Monitor Handlers (Part 1 of 4)

```

int main(void)
{
    _RFILE      *fp;
    int          i;
    volatile int  com;
    char         buf[RCD_LEN];
    char         cmd[100];
    /* Create a file. */
    sprintf(cmd, "CRTPF FILE(%s) RCDLEN(%d)", FILE_NAME, RCD_LEN);
    system(cmd);
    /* Open the file for write. */
    if ( (fp = _Ropen(FILE_NAME, "wr")) == NULL )
    {
        printf("Open for write fails\n");
        exit(1);
    }
    /* Write some data into the file. */
    memset(buf, '1', RCD_LEN);
    for ( i = 0; i < NUM_RCD; i++ )
    {
        _Rwrite(fp, buf, RCD_LEN);
    }
    _Rclose(fp);
    /* Open the file for the first read. */
    if ( (fp = _Ropen(FILE_NAME, "rr")) == NULL )
    {
        printf("Open for the first read fails\n");
        exit(2);
    }
    /* Read until end-of-file. */
    /* Since no signal handler or direct monitor handler is set up, */
    /* the EOF exception is ignored. The default value for SIGIO is */
    /* SIG_IGN. */
    i = 1;
    printf("The first read starts\n");
    while ( _Rreadn(fp, buf, RCD_LEN, __DFT)->num_bytes != EOF )
    {
        printf("Read record %d\n", i++);
    }
    _Rclose(fp);
    printf("The first read finishes\n");

    /* Set up a direct monitor handler and a signal handler. */
    /* Tell the direct monitor handler to handle the exception. */
    /* The direct monitor handler (exp_handler) calls the message */
    /* handler API QMHCHGEM with the parameter *HANDLE. This marks the */
    /* exception as handled. */
    /* Use exception classes to handle machine exceptions. */
    handle_flag = 1;
    #pragma exception_handler(exp_handler, com, 0, \
        _C2_MH_ESCAPE | _C2_MH_NOTIFY | _C2_MH_STATUS)
    signal(SIGIO, sig_handler);
    /* Open the file for the second read. */
    if ( (fp = _Ropen(FILE_NAME, "rr")) == NULL )
    {
        printf("Open for the second read fails\n");
        exit(3);
    }
}

```

Figure 133. T1520ICA — ILE C Source to Use Direct Monitor Handlers (Part 2 of 4)

```

/* Read until end of file. */
/* When the EOF exception is generated, the direct monitor handler */
/* is called first. Since it marks the exception as handled, */
/* the signal handler is not called. */
    i = 1;
    printf("The second read starts\n");
    while ( _Rreadn(fp, buf, RCD_LEN, __DFT)->num_bytes != EOF )
    {
        printf("Read record %d\n", i++);
    }
    _Rclose(fp);
    printf("The second read finishes\n");
/* Disable the direct monitor handler. */
    #pragma disable_handler
/* Set up a direct monitor handler and a signal handler. */
/* Set the global variable handle_flag to zero so that the */
/* direct monitor will not handle the exception. */
    handle_flag = 0;
    #pragma exception_handler(exp_handler, com, 0, \
        _C2_MH_ESCAPE | _C2_MH_NOTIFY | _C2_MH_STATUS)
    signal(SIGALL, sig_handler);
/* Open the file for the third read. */
    if ( (fp = _Ropen(FILE_NAME, "rr")) == NULL )
    {
        printf("Open for the third read fails\n");
        exit(4);
    }
/* Read until end-of-file. */
/* When the EOF exception is generated, the direct monitor handler */
/* is called first. Since the exception is not marked as */
/* handled, the signal handler is then called. */
    i = 1;
    printf("The third read starts\n");
    while ( _Rreadn(fp, buf, RCD_LEN, __DFT)->num_bytes != EOF )
    {
        printf("Read record %d\n", i++);
    }
    _Rclose(fp);
    printf("The third read finishes\n");

/* Disable the direct monitor handler. */
    #pragma disable_handler
/* Set up a direct monitor handler and a signal handler. */
    #pragma exception_handler(exp_handler, com, 0, \
        _C2_MH_ESCAPE | _C2_MH_NOTIFY | _C2_MH_STATUS)
    signal(SIGIO, sig_handler);
/* Open the file for the fourth read. */
    if ( (fp = _Ropen(FILE_NAME, "rr")) == NULL )
    {
        printf("Open for the fourth read fails\n");
        exit(5);
    }
/* Read until end-of-file. */
/* The EOF exception is generated in function read_file. Since */
/* there is no direct monitor handler for the read_file function, */
/* the signal handler is called. */

```

Figure 133. T1520ICA — ILE C Source to Use Direct Monitor Handlers (Part 3 of 4)


```

/* The direct monitor handler in main() is not called because the */
/* exception was mapped to SIGIO and the signal handler gets called */
/* at function read_file. */
    printf("The fourth read starts\n");
    read_file(fp);
    _Rclose(fp);
    printf("The fourth read finishes\n");
/* Disable the direct monitor handler. */
    #pragma disable_handler
}

```

Figure 133. T1520ICA — ILE C Source to Use Direct Monitor Handlers (Part 4 of 4)

2. To run the program T1520ICA, type:
CALL PGM(MYLIB/T1520ICA)

The first screen of output is shown below:

```

The first read starts
Read record 1
Read record 2
Read record 3
Read record 4
Read record 5
The first read finishes
The second read starts
Read record 1
Read record 2
Read record 3
Read record 4
Read record 5
In direct monitor handler
Exception message ID is CPF5001
The second read finishes
The third read starts
Read record 1

```

The second screen of output follows:

```

Read record 2
Read record 3
Read record 4
Read record 5
In direct monitor handler
Exception message ID is CPF5001
In signal handler
Exception message ID is CPF5001
The third read finishes
The fourth read starts
Read record 1
Read record 2
Read record 3
Read record 4
Read record 5
In signal handler
Exception message ID is CPF5001
The fourth read finishes
Press ENTER to end terminal session.
====>
F3=Exit F4=End of File F6=Print F9=Retrieve F17=Top
F18=Bottom F19=Left F20=Right F21=User Window

```

Nested Exceptions

Exceptions can be nested. A nested exception is an exception that occurs while another exception is being handled. When this happens, the processing of the first exception is temporarily suspended. Exception handling begins again with the most recently generated exception.

Note: If a nested exception causes the program to end, the exception handler for the first exception may not complete.

Example

The following example shows a nested exception.

```
#include <signal.h>
void hdlr_hdlr(_INTRPT_Hndlr_Parms_T * __ptr128 parms)
{
    /* Handle exception 2 using QMHCHGEM. */
}
void main_hdlr(_INTRPT_Hndlr_Parms_T * __ptr128 parms)
{
    #pragma exception_handler(hdlr_hdlr,0,0,_C2_MH_ESCAPE)
    /* Generate exception 2. */
    /* Handle exception 1 using QMHCHGEM. */
}
int main(void)
{
    #pragma exception_handler(main_hdlr,0,0,_C2_MH_ESCAPE)
    /* Generate exception 1. */
}
```

Figure 134. ILE C Source to Nest Exceptions

In this example, the `main()` function generates an exception which causes `main_hdlr` to get control. The handler `main_hdlr` generates another exception which causes `hdlr_hdlr` to get control. The handler `hdlr_hdlr` handles the exception. Control resumes in `main_hdlr`, and it handles the original exception.

As this example illustrates, you can get an exception within an exception handler. To prevent exception recursion, exception handler call stack entries act like control boundaries with regards to exception percolation. Therefore it is recommended that you monitor for exceptions within your exception handlers.

Using Cancel Handlers

Cancel handlers are used by C++ to call destructors during stack unwinding. It is recommended that you use the C++ try catch throw feature to ensure objects are destructed properly.

A cancel handler may be enabled around a body of code inside a function. When a cancel handler is enabled it only gets control if the suspend point of the call stack entry is inside that code (within the `#pragma cancel_handler` and `#pragma disable_handler` directives), and the call stack entry is canceled.

The `#pragma cancel_handler` directive provides a way to statically register a cancel handler within a call stack entry (or suspend point within a call stack entry). The Register Call Stack Entry Termination User Exit Procedure (CEERTX) and the Unregister Call Stack Entry Termination User Exit Procedure (CEETUTX) ILE

bindable APIs provide a way of dynamically registering a user-defined routine to be executed when the call stack entry for which it is registered is cancelled.

Cancel handlers provide an important function by allowing you to get control for clean-up and recovery actions when call stack entries are ended by something other than a normal return.

On the `#pragma cancel_handler` directive, the name of the cancel handler routine (a bound ILE procedure) is specified, along with a user-defined communications area. It is through the communications area that information is passed from the application to the handler function. When the cancel handler function is called, it is passed a pointer to a structure of type `_CNL_Hndlr_Parms_T` which is defined in the `<except.h>` header file. This structure contains a pointer to the communications area in addition to some other useful information that is passed by the system. This additional information includes the reason why the call was cancelled.

Example

The following simple example illustrates the use of the ILE Cancel Handler mechanism. This capability allows an application (program) the opportunity to have a user-provided function called to perform things such as error reporting/logging, closing of files, etc. when a particular function invocation is cancelled. The usual ways that cause cancellation to occur are: using the `exit()` function or the `abort()` function, using the `longjmp()` function to jump to an earlier call and having a CEE9901 Function Check generated from an unhandled exception.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <except.h>
/*-----*/
/* The following function is called a "cancel handler". It is      */
/* registered for a particular invocation (function) with the      */
/* #pragma cancel_handler directive. The variable identified      */
/* on this directive as the "communications area" can be accessed  */
/* using the 'Com_Area' member of the _CNL_Hndlr_Parms_T structure. */
/*                                                                */
/*-----*/
void CancelHandlerForReport( _CNL_Hndlr_Parms_T *cancel_info ) {
    printf("In Cancel Handler for function 'Report' ...\n");
    /* Changing the value in the communications area will update the */
    /* 'return_code' variable in the invocation being cancelled      */
    /* (in function 'Report' in this example). Note that the        */
    /* ILE C compiler will issue a warning for the following        */
    /* statement since it uses a non-ANSI C compliant technique.    */
    /* However, this will not affect the expected run-time behavior. */
    /* Set "return_code" in Report to an arbitrary number.          */
    *(volatile unsigned *)cancel_info->Com_Area = 500;
    printf("Communication Area now has the value: %d \n",
        *(volatile unsigned *)cancel_info->Com_Area );
    printf("Leaving Cancel Handler for function 'Report'...\n");
}
```

Figure 135. T1520XH4 — ILE C Source to Use Cancel Handlers (Part 1 of 3)

```

/*-----*/
/* The following function is also called a cancel handler but has */
/* been registered for the 'main' function. That is, when the */
/* 'main' function is cancelled, this function will automatically */
/* be called by the system. */
/* */
/*-----*/
void CancelHandlerForMain( _CNL_Hndlr_Parms_T *cancel_info ) {
    printf("In Cancel Handler for function 'main' ...\n");
    /* Changing the value in the communications area will update the */
    /* 'return_code' variable in the invocation being cancelled */
    /* (in function 'main' in this example). Note that the */
    /* ILE C compiler will issue a warning for the following */
    /* statement since it uses a non-ANSI C compliant technique. */
    /* However, this will not affect the expected run-time behavior. */
    /* Set "return_code" in main to an arbitrary number. */
    *( (volatile unsigned *)cancel_info->Com_Area ) = 999;
    printf("Communication Area now has the value: %d \n",
        *( (volatile unsigned *)cancel_info->Com_Area) );
    printf("Leaving Cancel Handler for function 'main'...\n");
}
/*-----*/
/* The following is simple function that registers another function */
/* (named 'CancelHandlerForReport' in this example) as its "cancel */
/* handler". When 'exit()' is used from this function, then this */
/* invocation and all prior invocations are cancelled by the system */
/* and any registered cancel handlers functions are automatically */
/* called. */
/*-----*/
void Report( void ) {
    volatile unsigned return_code; /* communications area */
    #pragma cancel_handler( CancelHandlerForReport, return_code )
    printf("in function Report()...about to call 'exit'...\n");
    /* Using the exit function will cause this function invocation */
    /* and all function invocations within this program to be */
    /* cancelled. If any of the functions being cancelled have */
    /* cancel handlers enabled, then those cancel handler functions */
    /* will be called by the system after each cancellation. */
    exit( 99 ); /* exit with an arbitrary value */
    printf("in function Report() just after calling 'exit'... \n");
    #pragma disable_handler
}
/*-----*/
/* In the 'main()' function a cancel handler is registered so that */
/* the function 'CancelHandlerForMain()' is called if 'main()' is */
/* cancelled. */
/*-----*/
int main( void ) {
    volatile unsigned return_code; /* communications area */
    #pragma cancel_handler( CancelHandlerForMain, return_code )
    return_code = 0; /* initialize return code which will */
    /* eventually be set in the cancel handler */
    printf("In main() about to call Report()...\n");
    Report();
    printf("...back from calling Report(). \n");
    printf("return_code = %d \n", return_code );
    #pragma disable_handler
}

```

Figure 135. T1520XH4 — ILE C Source to Use Cancel Handlers (Part 2 of 3)

```

/*-----*/
/* This program will result in the following screen output: */
/* */
/* In main() about to call Report()... */
/* in function Report()...about to call 'exit'... */
/* In Cancel Handler for function 'Report' ... */
/* Communication Area now has the value: 500 */
/* Leaving Cancel Handler for function 'Report'... */
/* In Cancel Handler for function 'main' ... */
/* Communication Area now has the value: 999 */
/* Leaving Cancel Handler for function 'main'... */
/*-----*/

```

Figure 135. T1520XH4 — ILE C Source to Use Cancel Handlers (Part 3 of 3)

Using Integrated Language Environment Condition Handlers

Integrated Language Environment (ILE) condition handlers allow you to register one or more condition handlers at run time. To register an ILE condition handler, use the Register ILE Condition Handler (CEEHDLR) bindable API. Include the <lecond.h> header file in your source code when using these APIs. ILE condition handlers may be un-registered by calling the Unregister ILE Condition Handler (CEEHDLU) bindable API.

Condition handlers are exception handlers that are registered at run time by using the Register ILE Condition Handler (CEEHDLR) bindable API. They are used to handle, percolate or promote exceptions. The exceptions are presented to the condition handlers in the form of an ILE condition.

Use the ILE bindable API CEEHDLR if you want to have a consistent mechanism of condition handling across several ILE languages; or for scoping exception handling to a call stack entry. CEEHDLR is scoped to the function that calls it, unlike the signal handler which is scoped to the activation group.

The ILE condition handler uses ILE conditions to allow greater cross-system consistency. An ILE condition is a system-independent representation of an error condition in an HLL.

The following example shows the source for a program MYPGM:

```

#include <lecond.h>
#include <stdio.h>
void my_handler(_FEEDBACK *cond, _POINTER *token, _INT4 *rc, _FEEDBACK *new );
void main_handler(_FEEDBACK *cond, _POINTER *token, _INT4 *rc, _FEEDBACK *new );
void fred(void)
{
    _HDLR_ENTRY hdlr=my_handler;
    char *p = NULL;
    CEEHDLR(&hdlr, NULL,NULL);
    *p = 'x';      /* exception */
    CEEHDLU(&hdlr,NULL);
}
int main(void)
{
    _HDLR_ENTRY hdlr=main_handler;
    CEEHDLR(&hdlr,NULL,NULL);
    fred();
}

```

Figure 136. T1520XH5 — ILE C Source to Use ILE Condition Handlers — main()

The following example shows the source for the service program HANDLERS:

```

#include <signal.h>
#include <stdio.h>
#include <lecond.h>
/* HANDLERS *SRVPGM (*CALLER) */
void my_handler(_FEEDBACK *cond, _POINTER *token, _INT4 *rc, _FEEDBACK *new)
{
    return;
}
void main_handler(_FEEDBACK *cond, _POINTER *token, _INT4 *rc, _FEEDBACK *new)
{
    printf("In main_handler\n");
}

```

Figure 137. T1520XH6 — ILE C Source to Use ILE Condition Handlers — Service Program

In the example, the procedure main() in MYPGM registers the condition handler main_handler followed by a call to the function fred() which registers the condition handler my_handler. Function fred() gets an exception causing my_handler to get control, followed by main_handler. The main() function is a control boundary. The exception is considered unhandled, so a function check is sent to function fred(). Handlers my_handler and main_handler are called again, this time for the function check. Neither of them handle the function check, so the program ends abnormally and CEE9901 is sent to the caller of the main() function.

The following example shows how to use a condition handler to handle an exception. In the example, ILE condition handler cond_hdlr is registered in the main() function using CEEHDLR. An MCH1211 (divide by zero) exception then occurs. Handler cond_hdlr is called and it indicates that the exception should be handled. Control then resumes in the main() function.

1. To create the program T1520IC6 using the source shown below, type:

```
CRTBNDC PGM(MYLIB/T1520IC6) SRCFILE(QCLE/QACSRC)
```

```

/* This program uses the ILE bindable API CEEHDLR to handle a      */
/* condition.                                                       */
#include <stdio.h>
#include <stdlib.h>
#include <leawi.h>
/* A condition handler registered by a call to CEEHDLR in main().    */
void cond_hdlr( _FEEDBACK *cond, _POINTER *token, _INT4 *rc, _FEEDBACK *new )
{
    *rc = CEE_HDLR_RESUME; /* handle the condition */
    printf("condition was raised: Facility_ID = %.3s, MsgNo = 0x%4.4x\n",
        cond->Facility_ID, cond->MsgNo);
}
int main(void)
{
    _HDLR_ENTRY hdlr = cond_hdlr;
    _FEEDBACK fc;
    int x,y;
    int zero = 0;
/* Register the condition handler.                                   */
    CEEHDLR(&hdlr, NULL, &fc);
    if (fc.MsgNo != CEE0000)
    {
        printf("Failed to register the condition handler\n");
        exit(88);
    }
/* Cause a divide by zero condition.                                */
    x = y / zero;
/* The code resumes here after the condition has been handled.      */
    printf("The condition was handled.\n");
}

```

Figure 138. T1520IC6 — ILE C Source to Use ILE Condition Handlers

2. To run the program T1520IC6 and receive the output shown below, type:

```
CALL PGM(MYLIB/T1520IC6)
```

```

condition was raised: Facility_ID = MCH, MsgNo = 0x1211
The condition was handled.
Press ENTER to end terminal session.

```

The next example shows the use of condition handlers. Using bindable API CEEHDLR, `main_hdlr` is registered in function `main()`, and `fred_hdlr` is registered in function `fred()`. An MCH1211 (divide by zero) exception occurs. Handler `fred_hdlr` is called to test if the exception is an MCH1211. The result code in the condition handler is set to percolate to the next condition handler. Handler `fred_hdlr` returns without handling the exception, causing `main_hdlr` to be called. The user-supplied token is updated to the value '1' and the result code is set to handle the exception. Handler `main_hdlr` returns, and the exception is handled. Control resumes in `fred()` following the statement that caused the divide by zero.

1. To create the program T1520IC7 using the following source, type:

```
CRTBND C PGM(MYLIB/T1520IC7) SRCFILE(QCLE/QACSRC)
```

```

/* This program uses the ILE bindable API CEEHDLR to enable handlers */
/* that percolate and handle a condition. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <leawi.h>
/* A condition handler registered by a call to CEEHDLR in fred(). */
void fred_hdlr( _FEEDBACK *cond, _POINTER *token, _INT4 *rc, _FEEDBACK *new )
{
    if (!memcmp(cond->Facility_ID, "MCH", 3) && cond->MsgNo == 0x1211)
    {
        *rc = CEE_HDLR_PERC; /* ... let it percolate to main ... */
        printf("in fred_hdlr, percolate exception.\n");
    }
    else
    {
        *rc = CEE_HDLR_RESUME; /* ... otherwise handle it. */
        printf("in fred_hdlr, handle exception.\n");
    }
}
/* A condition handler registered by a call to CEEHDLR in main(). */
void main_hdlr( _FEEDBACK *cond, _POINTER *token, _INT4 *rc, _FEEDBACK *new )
{
    printf("in main_hdlr: Facility_ID = %.3s, MsgNo = 0x%.4x\n",
        cond->Facility_ID, cond->MsgNo);
    **(_INT4 **)token = 1; /* Update the user's token. */
    *rc = CEE_HDLR_RESUME; /* Handle the condition */
}
int fred(void)
{
    _HDLR_ENTRY hdlr = fred_hdlr;
    _FEEDBACK fc;
    int x,y, zero = 0;
    /* Register the handler without a token. */
    CEEHDLR(&hdlr, NULL, &fc);
    if (fc.MsgNo != CEE0000)
    {
        printf("Failed to register the condition handler\n");
        exit(88);
    }
    /* Cause a divide by zero condition. */
    x = y / zero;
    printf("Resume here because resume cursor not moved and main_hdlr"
        " handled the exception\n");
}

int main(void)
{
    _HDLR_ENTRY hdlr = main_hdlr;
    _FEEDBACK fc;
    volatile _INT4 token=0, *tokenp = &token;
    /* Register the handler with a token of type _INT4. */
    CEEHDLR(&hdlr, (_POINTER *)&tokenp, &fc);
}

```

Figure 139. T1520IC7 — ILE C Source to Percolate a Message to Handle a Condition (Part 1 of 2)


```

    if (fc.MsgNo != CEE0000)
    {
        printf("Failed to register the condition handler\n");
        exit(99);
    }
    fred();
/* See if the condition handler for main() updated the token.          */
    if (*tokenp == 1)
        printf("A condition was percolated from fred() to main() and"
              " was then handled.\n");
}

```

Figure 139. T1520IC7 — ILE C Source to Percolate a Message to Handle a Condition (Part 2 of 2)

2. To run the program T1520IC7 and receive the output shown below, type:

```
CALL PGM(MYLIB/T1520IC7)
```

```

in fred_hdlr, percolate exception.
in main_hdlr: Facility_ID = MCH, MsgNo = 0x1211
Resume here because resume cursor not moved and main_hdlr handled the exception.
A condition was percolated from fred() to main() and was then handled.
Press ENTER to end terminal session.

```

The following example shows how to use condition handlers to promote an exception. Using the bindable API CEEHDLR, `main_hdlr` is registered in `main()`, and `fred_hdlr` is registered in `fred()`. An MCH1211 (divide by zero) exception then occurs. The handler `fred_hdlr` is called because of the MCH1211 exception. The handler `fred_hdlr` moves the resume cursor to the resume point in the `main()` function using the bindable API CEEMRCR. The handler `fred_hdlr` builds a condition token for CEE9902, and the result code is set to promote.

The handler `fred_hdlr` returns, and the original MCH1211 is promoted to a CEE9902. The handler `main_hdlr` is called because of the CEE9902 exception. The result code is set to handle the condition. The handler `main_hdlr` returns, and the CEE9902 is handled. Control resumes in the statement following the call to `fred()` in `main()`.

1. To create the program T1520IC8 using the source shown below, type:

```
CRTBNDC PGM(MYLIB/T1520IC8) SRCFILE(QCLE/QACSRC)
```

```

/* This program uses the ILE bindable API CEEHDLR to promote a          */
/* divide by zero condition to a CEE9902.                                */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <leawi.h>

```

Figure 140. T1520IC8 — ILE C Source to Promote a Message to Handle a Condition (Part 1 of 3)

```

/* A condition handler registered by a call to CEEHDLR in fred(). */
void fred_hdlr( _FEEDBACK *cond, _POINTER *token, _INT4 *rc, _FEEDBACK *new )
{
    _INT4 type=1;
    CEEMRCR(&type,NULL); /* Move the resume cursor to the resume point */
                          /* in main(). */
    /* If its a divide by zero error ... */
    printf("in fred_hdlr: moving resumes. ");
    printf("Facility_ID = %.3s, MsgNo = 0x%.4x\n",
           cond->Facility_ID, cond->MsgNo);
    if (!memcmp(cond->Facility_ID, "MCH", 3) && cond->MsgNo == 0x1211)
    {
        *rc = CEE_HDLR_PROM; /*... Promote the condition to unexpected error.*/
        *new = *cond;
        memcpy(new->Facility_ID, "CEE", 3);
        new->MsgNo = 9902;
        printf("promoting condition....\n");
    }
    else
    {
        *rc = CEE_HDLR_PERC; /*...Otherwise,Percolate to the next handler. */
        printf("percolating condition....\n");
    }
}

/* A condition handler registered by a call to CEEHDLR in main(). */
void main_hdlr( _FEEDBACK *cond, _POINTER *token, _INT4 *rc, _FEEDBACK *new )
{
    if (!memcmp(cond->Facility_ID,"CEE",3) && cond->MsgNo == 9902;)
        **(_INT4 **)token = 1; /* Got the promoted CEE9902. */
    else
        **(_INT4 **)token = 2; /* It is not a CEE9902. */
    *rc = CEE_HDLR_RESUME; /* Handle the condition. */
}

int fred(void)
{
    _HDLR_ENTRY hdlr = fred_hdlr;
    _FEEDBACK fc;
    int x,y, zero = 0;
    /* Register the handler without a token. */
    CEEHDLR(&hdlr, NULL, &fc);
    if (fc.MsgNo != CEE0000)
    {
        printf("Failed to register the condition handler\n");
        exit(88);
    }

    /* Cause a divide by zero condition. */
    x = y / zero;
    /* This is not the resume point because of the call to CEEMRCR in */
    /* fred_hdlr. */
    {
        printf("This is not the resume point: should not get here\n");
    }
}

```

Figure 140. T1520IC8 — ILE C Source to Promote a Message to Handle a Condition (Part 2 of 3)

```

int main(void)
{
    _HDLR_ENTRY hdlr = main_hdlr;
    _FEEDBACK fc;
    volatile _INT4 token=0, *tokenp = &token;
    /* Register the handler with a token of type _INT4. */
    CEEHDLR(&hdlr, (_POINTER *)&tokenp, &fc);
    if (fc.MsgNo != CEE0000)
    {
        printf("Failed to register the condition handler\n");
        exit(99);
    }
    fred();
    /* See if the condition handler for main() received the promoted */
    /* condition. */
    if (*tokenp == 1)
        printf("A condition was promoted from MCH1211 to CEE9902 by "
               "fred() and was handled by the condition handler enabled "
               "in main().\n");
}

```

Figure 140. T1520IC8 — ILE C Source to Promote a Message to Handle a Condition (Part 3 of 3)

2. To run the program T1520IC8 and receive the output shown below, type:

```
CALL PGM(MYLIB/T1520IC8)
```

```

in fred_hdlr: moving resumes. Facility_ID = MCH, MsgNo = 0x1211
promoting condition....
A condition was promoted from MCH1211 to CEE9902 by fred() and was handled by
the condition handler enabled in main().
Press ENTER to end terminal session.

```

Using the C/C++ signal Function to Handle Exceptions

HLL-specific handlers are the language features that are defined for handling errors. For ILE C/C++ `signal()` function can be used to handle exception messages.

iSeries system exceptions are mapped to C and C++ signals by the ILE C/C++ run-time environment. A signal handler determines the course of action for a signal. You cannot register a signal handler in an activation group that is different from the one you wish to call it from. If a signal handler is in a different activation group from the occurrence of the signal it is handling, the behavior is undefined.

Signals are raised implicitly or explicitly. To **explicitly** raise a signal, use the `raise()` function. Signals are **implicitly** raised by the iSeries system when an exception occurs. For example, if you call a program that does not exist, an implicit signal is raised indicating that the program object could not be found.

The header file `<signal.h>` contains a number of function prototypes that are associated with signal handling.

The following functions can be used with signal handling in your program:

- `raise()`
- `signal()`
- `_GetExcData()`

The `_GetExcData()` function is an ILE C/C++ extension that allows you to obtain information about the exception message associated with the signal and returns a structure containing information about the exception message. The `_GetExcData()` function returns the same structure that is passed to the `#pragma exception_handler` directive.

The `signal()` function specifies the action that is performed when a signal is raised. There are ten signals that are represented as macros in the `<signal.h>` header file. In addition, the macro `SIGALL` has the semantics of a signal but with some unique characteristics. The ten signals are as follows:

SIGABRT	Abnormal program end.
SIGFPE	Arithmetic operation error, such as dividing by zero.
SIGILL	An instruction that is not allowed.
SIGINT	System interrupt, such as receiving an interactive attention signal.
SIGIO	Record file error condition.
SIGOTHER	All other *ESCAPE and *STATUS messages that do not map to any other signals.
SIGSEGV	The access to storage is not valid.
SIGTERM	A end request is sent to the program.
SIGUSR1	Reserved for user-defined signal handler.
SIGUSR2	Reserved for user-defined signal handler.

SIG_IGN and **SIG_DFL** are signal actions that are also included in the `<signal.h>` header file.

SIG_IGN	Ignore the signal.
SIG_DFL	Default action for the signal.

`SIGALL` is an ILE C/C++ extension that allows you to register your own default-handling function for all signals whose action is `SIG_DFL`. This default-handling function can be registered by using the `signal()` function with `SIGALL`, as shown in the example section. A function check is not a signal and cannot be monitored for by the signal function. `SIGALL` cannot be signaled by the `raise()` function.

When a signal is received, the ILE C/C++ run-time environment handles the signal in one of three ways:

- If the value of the function is `SIG_IGN`, then the signal is ignored, because the exception is handled by the run-time environment and no signal handler is called. If the message that is mapped to the signal is an *ESCAPE or *NOTIFY message, then it is placed in the job log.
- If the value of the function is a pointer to a function, then the function that is addressed by the pointer is called.
- If the value of the function is `SIG_DFL`, then the system uses the value registered for `SIGALL` (choosing one of the three ways described here). If the value of the function for `SIGALL` is `SIG_DFL` then the exception is percolated.

Note: The value of the function is the function argument on the call to the `signal()` function.

A signal handling function is called when an exception occurs or when a signal is raised. A handler is defined with the `signal()` function. The value you assign on the `sig` parameter is associated with the function referred to on the `func` parameter. Properties of a signal handler are:

- When a signal handler is called in a C program, its corresponding signal action is set to `SIG_DFL`. You must reset the signal action if you want to handle the same signal again. If you do not, the default action is taken on subsequent exceptions. The handling of the signal can be reset from inside or outside the handler by calling `signal()`. For example:

```
signal ( SIGINT, &myhandler );
raise ( SIGINT );           /* signal is handled by myhandler.    */
...
raise ( SIGINT );           /* signal is handled by SIG_DFL.      */
...
signal ( SIGINT, &myhandler );/* reset signal handler to myhandler. */
raise ( SIGINT );           /* signal is handled by myhandler.    */
```

Figure 141. Resetting Signal Handlers

The default can be reset to `SIG_IGN`, another handler, or the same handler. You can recursively call the signal handler. Once stacked, multiple signal handler calls behave like any other calls. For example, if the action signal to the previous caller is chosen, the control will not be returned to the preceding caller (even if that call is another signal handler) but goes back to the previous caller.

- The `signal()` function returns the address of the previous signal handler for the specified signal, and sets the address of the new signal handler. You can stack the signal handlers yourself using the value returned by `signal()`. For example,

```
void (*func1) ();
void (*func2) ();
func1 = signal ( SIGINT, &handler2 ); /*func1 contains the address of */
                                     /*a previous signal handler or */
                                     /*SIG_DFL if no handler has been */
                                     /*defined.                      */
func2 = signal ( SIGINT, func1);      /*func2 contains the address of */
                                     /*handler2.                      */
```

Figure 142. Stacking Signal Handlers

Example

The following example shows how to set up a signal handler. The example illustrates that when there is no signal handler set up the default action for `SIGIO` is `SIG_IGN`. The exception is ignored. When a signal handler is set up for `SIGIO`, the signal handler is called.

1. To create the program `T1520SIG`, using the following source, type:

```
CRTBND C PGM(MYLIB/T1520SIG) SRCFILE(QCLE/QACSRC)
```

```

#include <stdio.h>
#include <signal.h>
#include <recio.h>
#include <stdlib.h>
#include <string.h>
#define FILE_NAME "QTEMP/MY_FILE"
#define RCD_LEN 80
#define NUM_RCD 5
/* The signal handler for SIGIO. */
static void handler_SIGIO(int sig)
{
    printf("In SIGIO handler\n");
    printf("Exception message ID is %7.7s\n", _EXCP_MSGID);
    printf("Signal raised is %d\n", sig);
}
/* The signal handler for SIGALL. */
static void handler_SIGALL(int sig)
{
    _INTRPT_Hndlr_Parms_T data;
    _GetExcData(&data);
    printf("In SIGALL handler\n");
    printf("Exception message ID is %7.7s\n", data.Msg_Id);
    printf("Signal raised is %d\n", sig);
}
int main(void)
{
    _RFILE *fp;
    int i;
    char buf[RCD_LEN];
    char cmd[100];
    /* Create a file. */
    sprintf(cmd, "CRTPF FILE(%s) RCDLEN(%d)", FILE_NAME, RCD_LEN);
    system(cmd);
    /* Open the file for write. */
    if ( (fp = _Ropen(FILE_NAME, "wr")) == NULL )
    {
        printf("Open for write fails\n");
        exit(1);
    }
    /* Write some data into the file. */
    memset(buf, '1', RCD_LEN);
    for ( i = 0; i < NUM_RCD; i++ )
    {
        _Rwrite(fp, buf, RCD_LEN);
    }
    _Rclose(fp);
    /* Open the file for the first read. */
    if ( (fp = _Ropen(FILE_NAME, "rr")) == NULL )
    {
        printf("Open for the first read fails\n");
        exit(2);
    }
    /* Read until end-of-file. */

```

Figure 143. T1520SIG — ILE C Source to Use Signal Handlers (Part 1 of 2)

```

/* Since there is no signal handler set up and the default      */
/* action for SIGIO is SIG_IGN, the EOF exception is ignored.  */
i = 1;
printf("The first read starts\n");
while ( _Rreadn(fp, buf, RCD_LEN, __DFT)->num_bytes != EOF )
{
    printf("Read record %d\n", i++);
}
_Rclose(fp);
printf("The first read finishes\n");
/* Set up a signal handler for SIGIO.                          */
signal(SIGIO, handler_SIGIO);
/* Open the file for the second read.                          */
if ( (fp = _Ropen(FILE_NAME, "rr")) == NULL )
{
    printf("Open for the second read fails\n");
    exit(3);
}
/* Read until end of file.                                     */
/* Since a signal handler is set up for SIGIO, the signal      */
/* handler is called when the EOF exception is generated.      */
i = 1;
printf("The second read starts\n");
while ( _Rreadn(fp, buf, RCD_LEN, __DFT)->num_bytes != EOF )
{
    printf("Read record %d\n", i++);
}
_Rclose(fp);
printf("The second read finishes\n");
/* Set up a signal handler for SIGALL.                          */
signal(SIGALL, handler_SIGALL);
/* Open the file for the third read.                            */
if ( (fp = _Ropen(FILE_NAME, "rr")) == NULL )
{
    printf("Open for the third read fails\n");
    exit(4);
}
/* Read until end of file.                                     */
/* Since there is no signal handler for SIGIO but there is a   */
/* signal handler for SIGALL, the signal handler for SIGALL    */
/* is called when the EOF exception is generated. But          */
/* the signal ID passed to the SIGALL signal handler is still  */
/* equal to SIGIO.                                             */
i = 1;
printf("The third read starts\n");
while ( _Rreadn(fp, buf, RCD_LEN, __DFT)->num_bytes != EOF )
{
    printf("Read record %d\n", i++);
}
_Rclose(fp);
printf("The third read finishes\n");
}

```

Figure 143. T1520SIG — ILE C Source to Use Signal Handlers (Part 2 of 2)

2. To run the program T1520SIG and receive the output shown below, type:
CALL PGM(MYLIB/T1520SIG)

```
The first read starts
Read record 1
Read record 2
Read record 3
Read record 4
Read record 5
```

```
The first read finishes
The second read starts
Read record 1
Read record 2
Read record 3
Read record 4
Read record 5
In SIGIO handler
Exception message ID is CPF5001
Signal raised is 9
The second read finishes
The third read starts
Read record 1
Read record 2
Read record 3
Read record 4
Read record 5
In SIGALL handler
Exception message ID is CPF5001
Signal raised is 9
The third read finishes
Press ENTER to end terminal session.
====>
F3=Exit F4=End of File F6=Print F9=Retrieve F17=Top
F18=Bottom F19=Left F20=Right F21=User Window
```

Control Boundary Examples for ILE C/C++

Use the following legend for the next three examples:

Activation Group	Dashed frame with rounded corners.
Program/Service Program	Dotted frame with square corners.
Call stack entry	Solid frame with rounded corners.
Control Boundary	A shaded call stack entry.

The example in Figure 144 on page 293 shows how a control boundary is defined in a simple ILE C/C++ application.

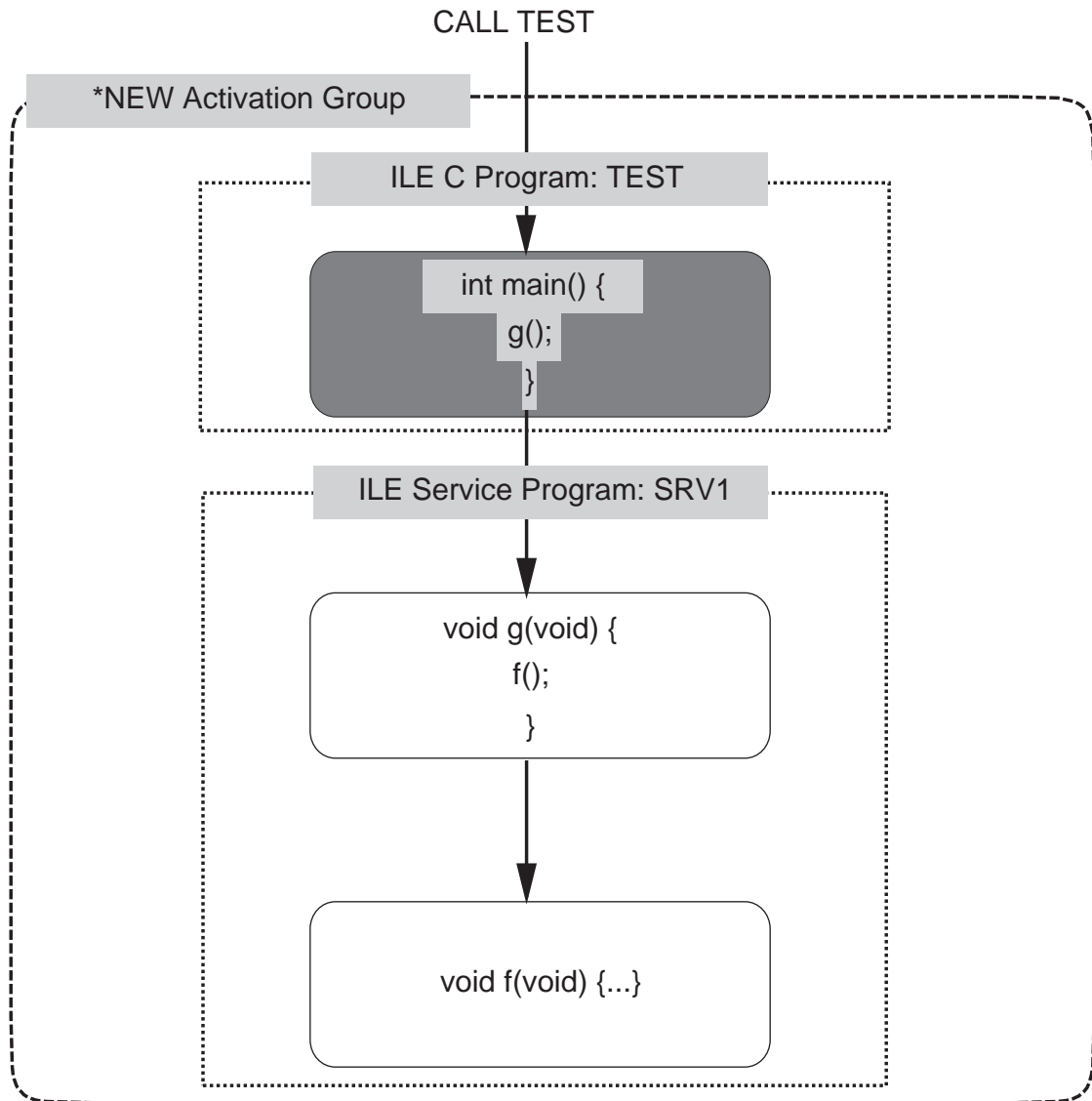


Figure 144. Example of Control Boundary When There is One Activation Group

This application runs in a single *NEW activation group (also called a system named activation group). The application consists of an ILE C program called TEST which is bound by reference to a service program SRV1. The service program SRV1 was created with the ACTGRP(*CALLER) option.

In this example, there is only one control boundary at procedure `main()` in program TEST. In reality the control boundary is at the PEP, but for simplification of the examples it will be considered to be at the procedure `main()`. This call stack entry is a control boundary since it is the first call stack entry in the activation group.

If an escape exception occurs in function `f()` and no handlers are enabled, then the exception percolates from the call stack entry for `f()` to the call stack entry for `g()`, and then to the call stack entry for `main()`. Since `main()` is the control boundary, and the exception is unhandled, it will be turned into a function check and be re-driven. The function check starts at call stack entry `f()` (where the exception occurred), and is percolated to the call stack entry for `g()`, and then to the call stack entry for `main()`. At this point, the function check has reached a control

boundary and was not handled. Therefore, a CEE9901 *ESCAPE message is sent to the caller of `main()`, and the activation group is ended.

The example in Figure 145 shows how control boundaries are set when calls are made between different activation groups.

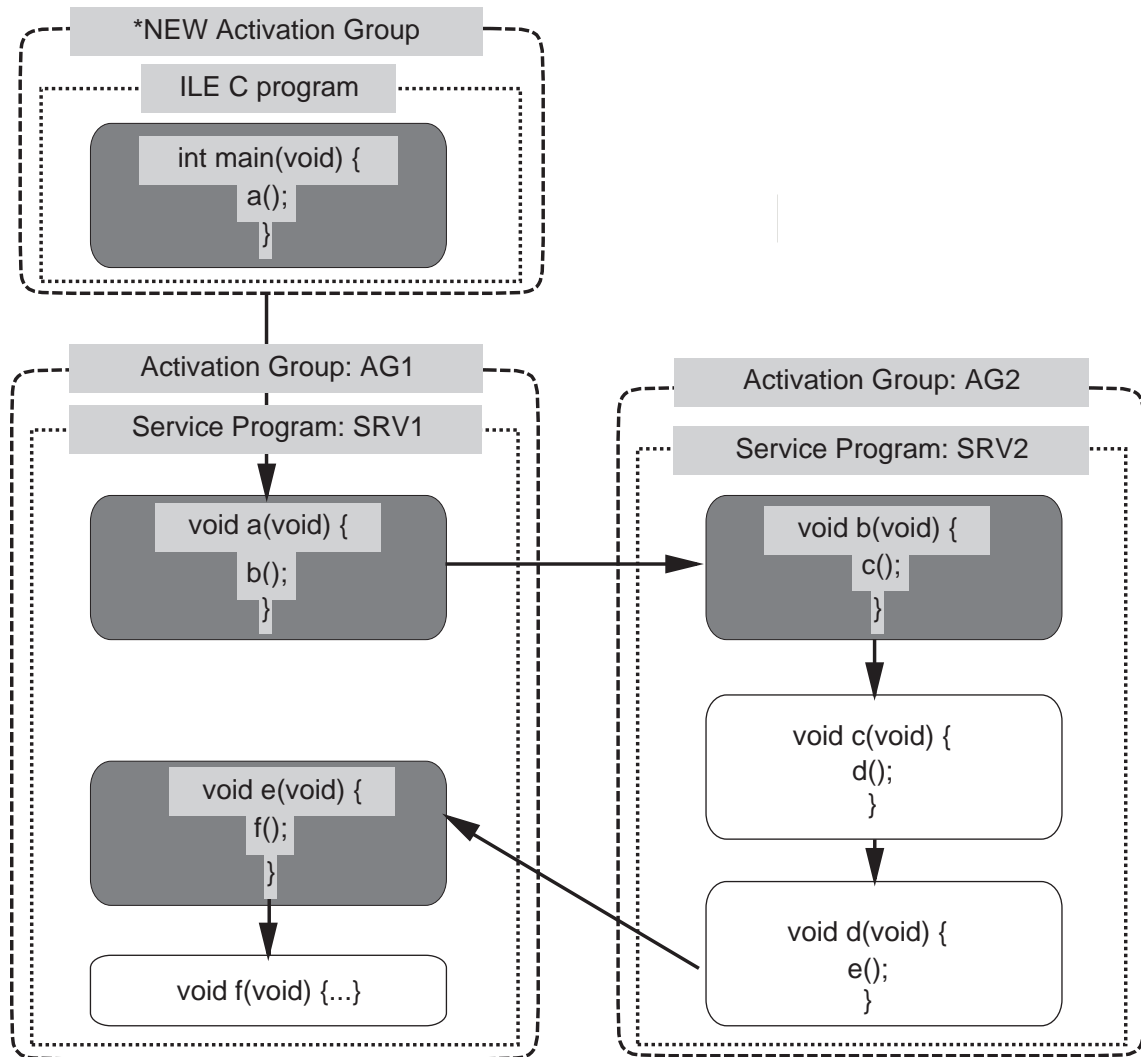


Figure 145. Example of Control Boundaries When You Have Multiple Activation Groups

This application consists of one ILE C/C++ program and two ILE C/C++ Service Programs (SRV1 and SRV2) that run in named activation groups (AG1 and AG2). In this example, the functions `main()`, `a()`, `b()`, and `e()` are all potential control boundaries. For example, when you are running in procedure `c()`, then `b()` would be the nearest control boundary.

The call stack entry for `main()` is a control boundary since it is the first call stack entry in the activation group. The call stack entries for `a()`, `b()`, and `e()` are control boundaries since the immediately preceding call stack entry for each runs in a different activation group.

If an escape exception occurs in function `d()` and no handlers are enabled, then the exception percolates from the call stack entry for `d()` to the call stack entry for `c()`, and then to the call stack entry for `b()`. Since the call stack entry `b()` is a control

boundary, the exception is turned into a function check and will be re-driven. The function check starts at function `d()`, and is percolated to the call stack entry for `c()`, and then to the call stack entry for `b()`. At this point, the function check has reached a control boundary and was not handled. Therefore, a CEE9901 *ESCAPE message is sent to the call stack entry for function `a()`, and the activation group AG2 is ended.

If `exit()` was called from within function `a()`, then it will cancel up the call stack entry for `a()`, control will return to the call stack entry for `main()`, and the activation group AG1 will be ended. At this time, any `atexit()` routines for activation group AG1 are called.

If `exit()` was called from within function `f()`, then it will cancel the call stack entries until it reaches the control boundary at the call stack entry for `e()` (for example, the call stack entries for both `e()` and `f()` are canceled). Since this control boundary is not the first one in activation group AG1, the activation group will NOT be taken down. Control will return to the call stack entry for function `d()`.

Note: If an `atexit()` was registered by function `f()`, it would not be called, since the activation group was not ended.

The example in Figure 146 on page 296 shows how control boundaries would be set for an application that is running in the default activation group. The application consists of two OPM programs A and C, and two ILE C programs B and D which were created using the `ACTGRP(*CALLER)` option on the `CRTPGM` command. In this example, the function `main()` in program B, and the function `main()` in program D are potential control boundaries.

The call stack entries for `main()` in programs B and D are control boundaries since the immediately preceding call stack entry for each is that of an OPM program.

If an escape exception occurs in function `g()` in program B and no handlers are enabled, then the exception percolates from the call stack entry for `g()` to the call stack entry for `main()`. Since the call stack entry for `main` is a control boundary, the exception is turned into a function check and will be re-driven. The function check starts at function `g()`, and is percolated to the call stack entry for `main()`. At this point, the function check has reached a control boundary and was not handled. The call stack entries for `g()` and `main` are cancelled, and a CEE9901 escape message is sent to the call stack entry for program A.

Note: The activation group will not be taken down, since the default activation group only goes away at the time the job ends.

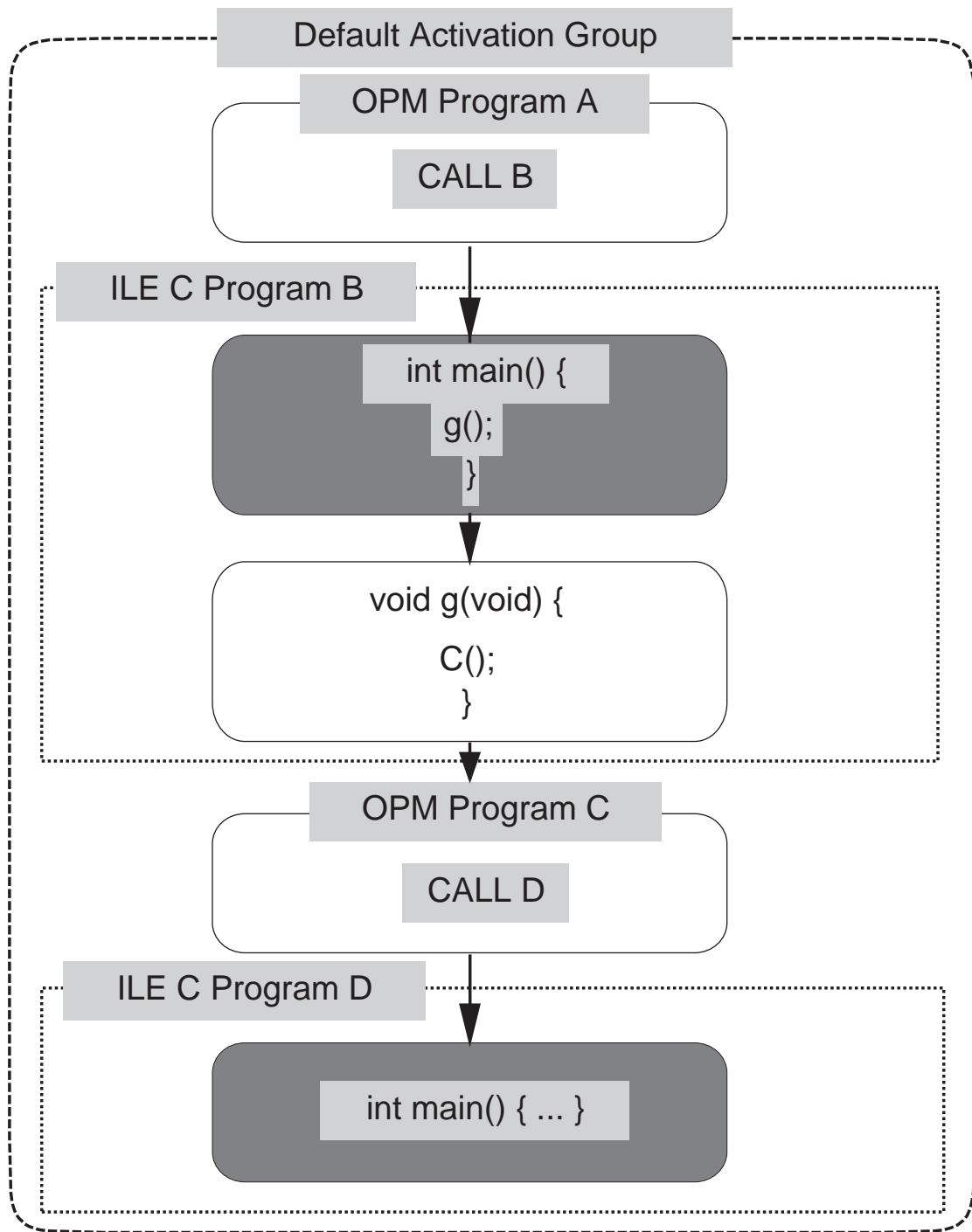


Figure 146. Example of Control Boundaries in the Default Activation Group

Exception Percolation: an Example

The following example is the sequence of exceptions and handling actions that occur when this source code is run:

```

#include <stdio.h>
#include <except.h>
#include <signal.h>
#include <lecond.h>
void handler1(_INTRPT_Hndlr_Parms_T * __ptr128 parms)
{
    printf("In handler1: will not handle the exception\n");
}
void handler2(_INTRPT_Hndlr_Parms_T * __ptr128 parms)
{
    printf("In handler2: will not handle the exception\n");
}
void handler3(_FEEDBACK *condition, _POINTER *token, _INT4 *result_code,
              _FEEDBACK *new_condition)
{
    printf("In handler3: will not handle the exception\n");
}
void handler4(_INTRPT_Hndlr_Parms_T * __ptr 128 parms)
{
    printf("In handler4: will not handle the exception\n");
}
void fred(void)
{
    _HDLR_ENTRY hdlr = handler3;
    char *p = NULL;
    #pragma exception_handler(handler2, 0, 0, \
                              _C2_MH_ESCAPE | _C2_MH_FUNCTION_CHECK)
    CEEHDLR(&hdlr, NULL, NULL);
    #pragma exception_handler(handler1, 0, 0, _C2_MH_ESCAPE)
    *p = 'x'; /* exception */
}
int main(void)
{
    signal(SIGSEGV, SIG_DFL);
    #pragma exception_handler(handler4, 0, 0, \
                              _C2_MH_ESCAPE | _C2_MH_FUNCTION_CHECK)
    fred();
}

```

Figure 147. T1520XH7 — ILE C Source for Exception Handling

1. An escape exception occurs in function `fred()`.
2. `handler1` gets control since it is monitoring for an escape message, it is the closest nested monitor to the exception, and it has highest priority since it is a direct handler.
3. `handler2` gets control since it is monitoring for an escape message, and has a higher priority than a CEEHDLR.
4. `handler3` gets control (from CEEHDLR).
5. `signal` handler gets control. Even though it is registered in `main`, `signal` is scoped to the activation group and therefore will get control. It gets control after `handler1`, `handler2`, and `handler3` since it has a lower priority than either direct handlers or CEEHDLRs. Since the action is `SIG_DFL`, the exception is not handled.
6. The exception is percolated to `main()`.
7. `handler4` gets control.
8. The exception is still not handled. Thus, when it hits the control boundary (the PEP for `main()`), it is turned into a function check and is re-driven.

9. handler1 does NOT get control, since it is not monitoring for a function check.
10. handler2 gets control since it is monitoring for function check.
11. handler3 gets control since CEEHDLRs get control for all *ESCAPE, *STATUS, *NOTIFY, and Function Check messages.
12. signal handler does NOT get control since signal does not recognize function checks.
13. The function check is percolated to main().
14. handler4 gets control since it is monitoring for function check.
15. The function check percolates to the control boundary and causes the ending.
16. (CEE9901) *ESCAPE is sent to the caller of main().

Chapter 14. Using iSeries Pointers in Your Program

This chapter describes how to:

- Use an open pointer
- Use other pointers
- Declare pointer variables
- Use pointer casting

In ANSI C, a **pointer** type is derived from a function type, a data object type, or an incomplete type. On the iSeries system, pointer types can also be derived from other iSeries entities such as system objects (for example, programs), code labels, and process objects. These pointer types are usually referred to as iSeries pointers and they are used extensively in the ILE C Machine Interface Library and in the ILE C/C++ exception handling structures and functions. The *ILE C/C++ for AS/400 MI Library Reference* contains information on ILE C Machine Interface Library Functions.

The iSeries pointer types are:

Open	Pointers that can hold any of the other pointer types.
Space	Generic pointers to data objects.
Function	System pointers to *PGM objects or procedure pointers to bound Integrated Language Environment procedures.
System	Pointers to system objects such as queues, indexes, libraries, and *PGM objects.
Label	Pointers to fixed locations within the executable code of a procedure or function.
Invocation	Pointers to process objects for procedure (function) calls under Integrated Language Environment, or program calls under EPM or OPM.
Suspend	Pointer to the location in a procedure where control has been suspended.

These pointer types, as well as pointers to data objects and incomplete types, are not data type (assignment or comparison) compatible with each other. For example, a variable declared as a pointer to a data object cannot be assigned the value of a function pointer or system pointer. A system pointer cannot be compared for equality with an invocation pointer or pointer to a data object. The above is not true for open pointers.

Note:

- Label pointers are only used by the **setjmp** macro.
- An open pointer is a pseudo-pointer type. It may contain any other pointer type, but it is not a pointer type unto itself.

Using Open Pointers

Before using open pointers in an ILE C/C++ program, consider the following constraints:

- An open pointer maps to a void pointer.
- An open (void) pointer can hold any type of pointer. It is data type compatible with all pointer types on the system. No compile-time error occurs when you cast an open pointer to other pointer types and when you cast other pointer types to an open pointer. You may receive a run-time exception if the pointer contains a value unsuitable for the context. For example, a system pointer in a pointer addition expression.
- An open pointer can be assigned to any type of pointer. You may receive a run-time exception if the type of pointer held in the open pointer is not data type compatible with the target of the assignment.
- An open pointer can be compared for equality (==, !=) to any pointer type.
- An open pointer can be compared in a relational operation (<, >, <=, >=) to another open pointer or to a data object pointer expression other than the NULL pointer. You may receive a run-time exception if the type of pointer that is held in the open pointer is not a pointer to a data object.

Note: Open pointers inhibit optimization. Use them only when absolutely necessary.

Using Pointers Other Than Open Pointers

Before you use pointers in an ILE C/C++ program, consider the following constraints:

- A NULL pointer can be assigned to and compared for equality (==, !=) with a pointer of any type.
- A NULL pointer cannot be used in a relational operation (<, >, <=, >=) with any pointer type.
- A pointer can be assigned to and compared for equality (==, !=) only with a pointer of the same type or an open pointer; otherwise a compile-time error occurs.
- Only pointers to data objects or open pointers that contain pointers to data objects can be used in relational operations (<, >, <=, >=); otherwise a compile-time error or run-time exception may occur.
- Only pointers to data objects can be used in arithmetic operations (+, -, ++, --); otherwise a compile-time error will occur.

Note: The conditional expression `if (!ptr)` is equivalent to the expression `if (ptr == NULL)`.

Declaring Pointer Variables

You can declare pointers to data and function pointers (pointers to bound ILE procedures) using the ILE C/C++ programming language. Pointers to *PGM objects (programs) can be declared in one of two ways:

1. Declare a pointer to a typedef that has been specified to have OS-linkage with the `#pragma linkage` directive. You must use `extern OS` for C++
2. Declare a system pointer (`_SYSPTR`).

You can declare variables of the other iSeries 400 pointer types by using the type definitions (typedefs) that are provided by the ILE C <pointer.h> header file.

Note: Pointers to OS-linkage functions (programs) and system pointers (_SYSPTR) are data type compatible. You can use a system pointer to hold the address of a program and call that program through the system pointer. However, a call through a system pointer that contains the address of a system object that is not a program results in undefined behavior.

These types are defined as pointers to void (void *), and the #pragma pointer directives in the header file cause the ILE C compiler to associate these types with the iSeries 400 pointer types.

Examples

The following example shows iSeries 400 pointer declarations:

```
#include <pointer.h>    /* The pointer header file.          */
_SYSPTR sysp;          /* A system pointer.          */
_SPCPTR spcp;          /* A space pointer.          */
_INVPTR invp;          /* An invocation pointer.    */
_OPENPTR opnp;         /* An open pointer.          */
_LBLPTR lblp;          /* A label pointer.          */
void (*fp) (int);      /* A function pointer.       */
#pragma datamodel (p128)
#pragma linkage (OS_FN_T, OS)
#pragma datamodel (pop)
typedef void (OS_FN_T) (int); /* Typedef of an OS-linkage function.*/
OS_FN_T * os_fn_p;      /* An OS-linkage function pointer.  */
int * ip;               /* A pointer to a data object.    */
```

Figure 148. ILE C Source to Declare Pointer Variables

```
#include <pointer.h>    /* The pointer header file.          */
_SYSPTR sysp;          /* A system pointer.          */
_SPCPTR spcp;          /* A space pointer.          */
_INVPTR invp;          /* An invocation pointer.    */
_OPENPTR opnp;         /* An open pointer.          */
_LBLPTR lblp;          /* A label pointer.          */
void (*fp) (int);      /* A function pointer.       */
#pragma datamodel (p128)
#pragma linkage (OS_FN_T, OS)
#pragma datamodel (pop)
typedef void (OS_FN_T) (int); /* Typedef of an OS-linkage function.*/
OS_FN_T * os_fn_p;      /* An OS-linkage function pointer.  */
int * ip;               /* A pointer to a data object.    */
```

Figure 149. ILE C++ Source to Declare Pointer Variables

A function pointer is a pointer that points to either a bound procedure (function) within an Integrated Language Environment program object, or an OS-linkage program object (system pointer) in the system.

The following example shows you how to declare a pointer to a bound procedure (a function that is defined within the same ILE program object):

```

int fct1( void ) {...}
int fct2( void ) {...}
int (*fct_ptr)(void) = fct1;
int main()
{
    fct_ptr();          /* Call fct1() using fct_ptr.      */
    fct_ptr = fct2;     /* Dynamically set fct_ptr to fct2.*/
    fct_ptr();          /* Call fct2() using fct_ptr.      */
}

```

Figure 150. ILE C Source to Declare a Pointer to a Bound Procedure

The following example shows you how to declare a pointer to an iSeries 400 program as a function pointer with OS-linkage. If the `#pragma linkage OS` directive is omitted from the code, the ILE C compiler assumes that `os_fct_ptr` is a pointer to a bound C function returning void, and will issue a compile error for incompatible pointer types between `os_fct_ptr` and the system pointer returned by `rslvsp()`.

```

#include <miptrnam.h>
#include <stdio.h>
#pragma datamodel(p128)
typedef void (OS_fct_t) ( void );
#pragma linkage(OS_fct_t,OS)
#pragma datamodel(pop)
int main ( void )
{
    OS_fct_t *os_fct_ptr;
    char    pgm_name[10];
    printf("Enter the program name : \n");
    scanf("%s", pgm_name);
    /* Dynamic assignment of a system pointer to program "MYPGM" */
    /* in *LIBL. The rslvsp MI library function will resolve to */
    /* this program at runtime and return a system pointer to */
    /* the program object. */
    os_fct_ptr = rslvsp(_Program, pgm_name, "*LIBL", _AUTH_OBJ_MGMT);
    os_fct_ptr();          /* OS-linkage *PGM call using a */
                          /* pointer. */
}

```

Figure 151. ILE C Source to Declare a Pointer to an iSeries 400 Program as a Function Pointer

```

#include <miptrnam.h>
#include <stdio.h>
typedef void (OS_fct_t) ( void );
#ifdef __cplusplus
extern OS { void (OS_FN_T) (int); }
#else
#pragma linkage (OS_FN_T, OS)
typedef void(OS_FN_T) (int);
#endif
int main ( void )
{
    OS_fct_t *os_fct_ptr;
    char      pgm_name[10];
    printf("Enter the program name : \n");
    scanf("%s", pgm_name);
    /* Dynamic assignment of a system pointer to program "MYPGM" */
    /* in *LIBL. The rslvsp MI library function will resolve to */
    /* this program at runtime and return a system pointer to */
    /* the program object. */
    os_fct_ptr = rslvsp(_Program, pgm_name, "*LIBL", _AUTH_OBJ_MGMT);
    os_fct_ptr();          /* OS-linkage *PGM call using a */
                          /* pointer. */
}

```

Figure 152. ILE C++ Source to Declare a Pointer to an iSeries Program as a Function Pointer

Using Pointer Casting

In the C language, **casting** is a construct to view a data object temporarily as another data type. There are several constraints when using pointer casting, especially for non-data object pointers. They are as follows:

- You can cast a pointer to another pointer of the same iSeries pointer type. If the ILE C compiler detects a type mismatch in an expression, a compile-time error occurs.
- An open (void) pointer can hold a pointer of any type. Casting an open pointer to other pointer types and casting other pointer types to an open pointer does not result in a compile-time error. You may receive a run-time exception if the pointer contains a value unsuitable for the context.
- When you convert a valid data object pointer to a signed or unsigned integer type, the return value is the offset of the pointer. If the pointer is NULL, the conversion returns a value of zero (0). It is not possible to determine whether the conversion originated from a NULL pointer or a valid pointer with an offset 0.
- When you convert a valid function (procedure) pointer, system pointer, invocation pointer, label pointer, or suspend pointer to a signed or unsigned integer type, the result is always zero.
- When you convert an open pointer that contains a valid space address, the return value is the offset that is contained in the address.
- You can convert an integer to pointer, but the resulting pointer value cannot be dereferenced. The right four bytes of such a pointer will contain the original integer value, and this value can be recovered by converting the pointer back to

an integer. This marks a change from behavior exhibited in earlier versions of ILE C, where integer to pointer conversions always resulted in a NULL pointer value.

The following example shows iSeries pointer casting:

```
#include <pointer.h>
#pragma datamodel(p128)
#pragma linkage(TESTPTR, OS)
#pragma datamodel(pop)
void TESTPTR(void);      /* System pointer to this program */
_SYSPTR sysp;           /* System pointer */
_OPENPTR opnp;          /* open pointer */
void (*fp)(void);       /* function pointer */
int i = 1;              /* integer */
int *ip = &i;           /* Space pointer */
void main (void) {
    fp = &main;         /* initialize function pointer */
    sysp = &TESTPTR;    /* initialize system pointer */

    i = (int) ip;        /* segment offset stored in i */
    ip = (int *) i;      /* address stored is invalid */
    i = (int) fp;        /* zero is stored in i */
    i = 2;
    fp = (void (*)( )) i; /* address stored is invalid */
    i = 3;
    sysp = (_SYSPTR) i;  /* address stored is invalid */
    opnp = &i;          /* address of i stored in open pointer */
    i = (int) opnp;      /* offset of space pointer contained */
                        /* in open pointer is stored in i */
    i = 4;
    opnp = (_OPENPTR) i; /* address stored is invalid */
    i = (int) opnp;      /* i is set to integer value stored (4)*/
}
```

Figure 153. ILE C Source to Show iSeries Pointer Casting

The following example demonstrates how to pass iSeries pointers as arguments on a dynamic program (OS-linkage) call to another ILE C program.

The example consists of 2 ILE C programs. Program 1 passes several types of iSeries pointers as arguments to Program 2. Program 2 receives the arguments and checks them to make sure that they were passed correctly.

1. To create the program T1520DL8 using the following source, type:

```
CRTBNDC PGM(MYLIB/T1520DL8) SRCFILE(QCLE/QACSRC)
```

```

/* This program passes several types of iSeries pointers as arguments */
/* to another ILE C program T1520DL9.                                     */
#include <stdio.h>
#include <pointer.h>
#pragma(pl28)
typedef struct {
    _SPCPTR spcptr;    /* A space pointer.          */
    _SYSPTR sysptr;    /* A system pointer.         */
    void (*fnptr)();   /* A function pointer.       */
} PtrStructure;
#pragma linkage (T1520DL9, OS)

```

Figure 154. T1520DL8 — ILE C Source that Uses iSeries Pointers (Part 1 of 2)

```

#pragma datamodel(pop)
void T1520DL9 (PtrStructure *, _SPCPTR, _SYSPTR, void (*)());
void function1(void) /* A function definition.          */
{
    printf("Hello!\n");
}

int main(void)
{
    int          i = 4;
    PtrStructure ptr_struct;
    /* Make assignments to the fields of ptr_struct.          */
    ptr_struct.spcptr = (_SPCPTR)&i; /* A space pointer.          */
    ptr_struct.sysptr = (_SYSPTR)T1520DL9; /* A system pointer.        */
    ptr_struct.fnptr = &function1; /* A function pointer.       */

    /* Call T1520DL9, passing the address of ptr_struct and other */
    /* valid iSeries pointer arguments.                             */

    T1520DL9(&ptr_struct, (_SPCPTR)&i, (_SYSPTR)T1520DL9, &function1);
}

```

Figure 154. T1520DL8 — ILE C Source that Uses iSeries Pointers (Part 2 of 2)

2. To create the program T1520DL9 using the following source, type:
 CRTBNDC PGM(MYLIB/T1520DL9) SRCFILE(QCLE/QACSRC)

```

/* This program receives the arguments from T1520DL8 and checks them */
/* to make sure they were passed correctly. */
#include <stdio.h>
#include <pointer.h>
typedef struct {
    _SPCPTR spcptr; /* A space pointer. */
    _SYSPTR sysptr; /* A system pointer. */
    void (*fnptr)(); /* A function pointer. */
} PtrStructure;
int main( int argc, char **argv)
{
    _OPENPTR openptr; /* An open pointer. */
    _SPCPTR spcptr; /* A space pointer. */
    _SYSPTR sysptr; /* A system pointer. */
    void (*fnptr)(); /* A function pointer. */
    PtrStructure *ptr_struct_ptr;
    int error_count = 0;
    /* Receive the structure pointer passed into a local variable. */
    ptr_struct_ptr = (PtrStructure *)argv[1];

```

Figure 155. T1520DL9 — ILE C Source that Uses iSeries Pointers (Part 1 of 2)

```

/* Receive the iSeries pointers passed into an open pointer, */
/* then assign them to pointers of their own type. */
openptr = (_OPENPTR)argv[2];
spcptr = openptr; /* A space pointer. */

openptr = (_OPENPTR)argv[3];
sysptr = openptr; /* A system pointer. */

openptr = (_OPENPTR)argv[4];
fnptr = openptr; /* A function pointer. */

/* Test the pointers passed with the pointers in ptr_struct_ptr. */

if (spcptr != ptr_struct_ptr->spcptr)
    ++error_count;
if (sysptr != ptr_struct_ptr->sysptr)
    ++error_count;
if (fnptr != ptr_struct_ptr->fnptr)
    ++error_count;

if (error_count > 0)
    printf("Pointers not passed correctly.\n");
else
    printf("Pointers passed correctly.\n");
return;
}

```

Figure 155. T1520DL9 — ILE C Source that Uses iSeries Pointers (Part 2 of 2)

3. To run the program T1520DL8, type:

```
CALL PGM(MYLIB/T1520DL8)
```

The output is as follows:

Pointers passed correctly.
Press ENTER to end terminal session.

Chapter 15. Using Packed Decimal Data in Your C Programs



This chapter describes how to:

- Convert from packed decimal data types
- Pass a pointer to packed decimal data to a function
- Call another program that contains packed decimal data
- Use library functions with packed decimal data
- Understand packed decimal data type errors

The ILE C compiler supports the packed decimal data type, as an extension to ANSI C. This is strictly a C data type. C++ decimal support is provided in a class. For more information, refer to *ILE C/C++ Language Reference*.

You can use the packed decimal data type to represent large numeric quantities accurately, especially in business and commercial applications for financial calculations. For example, the fractional part of a dollar can be represented accurately by two digits that follow the decimal point. You do not have to use floating point arithmetic which is more suitable for scientific and engineering computations (which often use numbers that are much larger than the largest packed decimal variable can store).

The packed decimal data type allows representation of up to 31 significant digits, including integral, and fractional parts. You can declare typedefs, arrays, structures, and unions that have packed decimal members. You can apply operators (unary operators) on packed decimal variables. Bitwise operators do not apply to packed decimal data. The packed decimal data type in ILE C is compatible with packed decimal representations in RPG for iSeries and COBOL for iSeries. You can also define macros and call library functions with packed decimal arguments. *ILE C for AS/400 Language Reference* contains information on the packed decimal data type.

Note: To use the `decimal`, `digitsof`, and `precisionof` macros in your code you must specify the `<decimal.h>` header file in your ILE C source.

Converting from Packed Decimal Data Types

If the value of the packed decimal type to be converted is within the range of values that can be represented exactly, the value of the packed decimal type is not changed. Packed decimal types are compatible if their types are the same. For example, `decimal(n_1 , p_1)` and `decimal(n_2 , p_2)` have compatible types if and only if $((n_1 = n_2) \text{ and } (p_1 = p_2))$.

Converting from a Packed Decimal Type to a Packed Decimal Type

The following example illustrates different conversions from packed decimal types to packed decimal types that have different sizes. If the value of the packed decimal type to be converted is not within the range of values that can be represented exactly, the value of the packed decimal type is truncated. If truncation occurs in the fractional part, the result is truncated, and there is no run-time error.

```

#include <decimal.h>
int main (void)
{
    decimal(4,2) targ_1, targ_2;
    decimal(6,2) op_1=1234.56d, op_2=12.34d;
    targ_1=op_1;      /* A run-time error is generated because the integral
                       part is truncated; targ_1=34.56d.          */
    targ_2=op_2;      /* No run-time error is generated because neither the
                       integral nor the fractional part is truncated;
                       targ_2=12.34d.                            */
}

```

Figure 156. ILE C Source to Convert Packed Decimals

If assignment causes truncation in the integral part, then there is a run-time error. A run-time exception occurs when an integral value is lost during conversion to a different type, regardless of what operation requires the conversion. See “Understanding Packed Decimal Data Type Errors” on page 320 for an example of run-time exceptions.

Examples

There is no warning or error during compilation on assignment to a smaller target. See “Understanding Packed Decimal Data Type Errors” on page 320 for information on compile-time and run-time errors during conversion.

The following example shows conversion from one packed decimal type to another with a smaller precision. Truncation on the fractional part results.

```

#include <decimal.h>
int main(void)
{
    decimal(7,4) x = 123.4567D;
    decimal(7,1) y;
    y = x;      /* y = 123.4D */
}

```

Figure 157. ILE C Source to Convert a Packed Decimal to a Packed Decimal with Smaller Precision

The next example shows conversion from one packed decimal type to another with a smaller integral part. Truncation on the integral part results. The `#pragma nosigtrunc` directive turns off exceptions generated because of overflow.

```

#pragma nosigtrunc
#include <decimal.h>
int main (void)
{
    decimal(8,2) x = 123456.78D;
    decimal(5,2) y;
    y = x;      /* y = 456.78D */
}

```

Figure 158. ILE C Source to Convert a Packed Decimal to a Packed Decimal with Smaller Integral Part

The next example shows conversion from one packed decimal type to another with a smaller integral part and smaller precision. Truncation on both integral and fractional parts results. The `#pragma nosigtrunc` directive turns off exceptions generated because of overflow.

```
#pragma nosigtrunc
#include <decimal.h>
int main (void)
{
    decimal(8,2) x = 123456.78D;
    decimal(4,1) y;
    y = x; /* y = 456.7D */
}
```

Figure 159. ILE C Source to Convert a Packed Decimal to a Packed Decimal with Smaller Integral Part and Smaller Precision

Converting from a Packed Decimal Type to an Integer Type

When you convert a value of a packed decimal type to an integer type, the value becomes a packed decimal(20,0), which then becomes an integer type. High-order bits will be truncated depending on the size of the integer type. No run-time exception occurs when assigning a packed decimal to an integer type that results in truncation of the integral part.

Examples

The following example shows the conversion from a packed decimal type that has a fractional part to an integer type.

```
#include <decimal.h>
int main (void)
{
    int op;
    decimal(7,2) op1 = 12345.67d;
    op = op1; /* Truncation on the fractional */
             /* part. op=12345 */
}
```

Figure 160. ILE C Source to Convert a Packed Decimal with a Fractional Part to an Integer

The following example shows the conversion from a packed decimal type that has less than 10 digits in the integral part to an integer type.

```
#include <decimal.h>
int main(void)
{
    int op;
    decimal(3) op2=123d;
    op = op2; /* No truncation and op=123 */
}
```

Figure 161. ILE C Source to Convert a Packed Decimal with Less than 10 Digits in the Integral Part to an Integer

The following example shows the conversion from a packed decimal type that has more than 10 digits in the integral part to an integer type.

```

#include <decimal.h>
int main (void)
{
    int op2;
    decimal(12) op3;
    op3 = 123456789012d;
    op2 = op3;                                /* High-order bits will be truncated.*/
                                              /* op2 = 0xBE991A14          */
}

```

Figure 162. ILE C Source to Convert a Packed Decimal with More than 10 Digits in the Integral Part to an Integer

The following example shows conversion from a packed decimal type that has a fractional part, and an integral part having more than 10 digits to an integer type.

```

#include <decimal.h>
int main (void)
{
    int op;
    long long op_2;
    decimal(15,2) op_1 = 1234567890123.12d;
    op = op_1;                                /* High-order bits will be truncated. */
    op_2 = op_1;                               /* op_2 = 1234567890123, op = 0x71FB04CB */
}

```

Figure 163. ILE C Source to Convert a Packed Decimal with More than 10 Digits in Both Parts to an Integer

Converting from a Packed Decimal Type to a Floating Point Type

When a value of packed decimal type is converted to floating type, if the value being converted is outside the range of values that can be represented, then the behavior is undefined. If the value being converted is within the range of values that can be represented, but cannot be represented exactly, the result is truncated. When a float or a double is converted to a packed decimal with smaller precision, the fractional part of the float or the double will be truncated.

The following example shows the conversion from a packed decimal type to a floating point type.

```

#include <decimal.h>
#include <stdio.h>
int main(void)
{
    decimal(5,2) dec_1=123.45d;
    decimal(11,5) dec_2=-123456.12345d;
    float f1,f2;
    f1=dec_1;
    f2=dec_2;
    printf("f1=%f\nf2=%f\n\n",f1,f2);      /* f1=123.449997      */
                                              /* f2=-123456.125000 */
}

```

Figure 164. ILE C Source to Convert a Packed Decimal to a Floating Point

The output is as follows:

```
f1=123.449997
f2=-123456.125000
Press ENTER to end terminal session.
```

Overflow Behavior

The following table describes the overflow behavior when a packed decimal number is assigned to a smaller target. An exception is not generated when:

- A packed decimal is assigned to a smaller target with integral type.
- A packed decimal is assigned to a smaller target of floating point type.

An exception is generated when a packed decimal is assigned to a smaller packed decimal target. You can suppress run-time errors by using the `#pragma nosigtrunc` directive in your ILE C source code.

Table 14. Handling Overflow From a Packed Decimal to a Smaller Target

From Field	To Field	Run-Time Error
Packed Decimal	char, int, short, long, long long, bit	No
Packed Decimal	Packed Decimal	Yes
Packed Decimal	Float	No ¹
Packed Decimal	Double	No ¹

Note: ¹ There is no packed decimal number large enough to cause overflow when the packed decimal is assigned to a float or double. If you use the MI instruction set `ca` to unmask a floating point exception, you receive an error message MCH1213 for a floating point inexact result.

Passing Packed Decimal Data to a Function

There are no default argument promotions on arguments that have packed decimal type when the called function does not include a prototype. This means that any function definition that contains packed decimal arguments has to be prototyped. Otherwise the behavior is undefined. The argument with packed decimal type is aligned on 4, 8, or 16 byte boundaries depending on the size of the packed decimal type. For example:

- $1 \leq n \leq 7$ aligns on a 4-byte boundary
- $8 \leq n \leq 15$ aligns on a 8-byte boundary
- $16 \leq n \leq 31$ aligns on a 16-byte boundary

The following example shows how to pass packed decimal variables to a function.

```
#include <decimal.h>
#include <stdio.h>
decimal(3,1)    d1 = 33.3d;
decimal(10,5)   d2 = 55555.55555d;
decimal(28)     d3 = 88888888888888888888888888888888d;
void func1( decimal(3,1), decimal(10,5),
            decimal(10,5), decimal(28));
```

Figure 165. ILE C Source to Pass Packed Decimal Variable to a Function (Part 1 of 2)

Figure 165. ILE C Source to Pass Packed Decimal Variable to a Function (Part 2 of 2)

```
x1 = 33.3  
x2 = 5555.5555  
x3 = 999.99  
x4 = 88888888888888888888888888888888
```

The following example shows you how to pass a pointer to a packed decimal variable to a function.

Figure 166. ILE C Source to Pass a Pointer to a Packed Decimal Value to a Function

The packed decimal argument in the function call has to be the same type as the packed decimal in the function prototype. If overflow occurs in a function call with packed decimal arguments, no error or warning is issued during compilation, and a run-time exception is generated.

The output is as follows:

```
The packed decimal number is: 123.45
Press ENTER to end terminal session.
```

Calling Another Program that Contains Packed Decimal Data

You can pass packed decimal arguments with interlanguage calls to RPG or COBOL.

Example

The following example shows an ILE C program that calls an OPM COBOL program and then passes a packed decimal data.

```
#include<stdio.h>
#include <decimal.h>
void CBLPGM(decimal(9,7));
#pragma datamodel(p128)
#pragma linkage(CBLPGM,OS)
#pragma datamodel(pop)
int main(void)
{
    decimal(9,7) arg=12.1234567d;
    /* Call an OPM COBOL program and pass a packed          */
    /* decimal argument to it.                               */
    CBLPGM(arg);
    printf("The COBOL program was called and passed a packed decimal value\n");
}
```

Figure 167. ILE C Source for an ILE C Program that Passes Packed Decimal Data

The following example shows COBOL source for passing a packed decimal variable to an ILE C program.

```

IDENTIFICATION DIVISION.
PROGRAM-ID. CBLPGM.
*****
*   Packed decimals:  This is going to be called by an ILE C   *
*   program to pass packed decimal data.                       *
*****
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-AS400.
OBJECT-COMPUTER. IBM-AS400.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
DATA DIVISION.
FILE SECTION.
WORKING-STORAGE SECTION.
77  PAC-DATA                      PIC X(30) VALUE "PROGRAM START".
77  PACK-IN-WS                   PIC 99.9999999.
LINKAGE SECTION.
01  PACK-DATA                     PIC 9(2)V9(7) PACKED-DECIMAL.
PROCEDURE DIVISION USING PACK-DATA.
MAIN-LINE SECTION.
    MOVE PACK-DATA TO PACK-IN-WS.
    DISPLAY "****  PACKED DECIMAL RECEIVED IS: " PACK-IN-WS.
    GOBACK.

```

Figure 168. COBOL Source that Receives Packed Decimal Data from an ILE C Program

The output is as follows:

```

**** PACKED DECIMAL RECEIVED IS: 12.1234567
Press ENTER to end terminal session.

```

```

The COBOL program was called and passed a packed decimal value.
Press ENTER to end terminal session.

```

Using Library Functions with a Packed Decimal Data Type

You can use the `va_arg` macro to accept a packed decimal data of the form (n,p) . You can write packed decimal constants to a file, and scan them back.

Examples

The following example shows you how to use the `va_arg` macro to accept a packed decimal data of the form (n,p) . The `va_arg` macro returns the current packed decimal argument.


```

/* This program uses the va_arg macro to accept a static decimal    */
/* data type of the form decimal(n,p). The va_arg macro returns    */
/* the current packed decimal argument.                            */
#include <decimal.h>
#include <stdio.h>
#include <stdarg.h>
#define N1 3
#define N2 6
int vargf(FILE *,char *,...);
int main(void)
{
    int num1 = -1, num2 = -1;
    char fmt_1[]="%D(3,2)%D(3,2)%D(3,2)";
    char fmt_2[]="%D(3,2)%D(3,2)%D(3,2)%D(3,2)%D(3,2)%D(3,2)";
    decimal(3,2) arr_1[]={ 1.11d, -2.22d, 3.33d };
    decimal(3,2) arr_2[]={ -1.11d, 2.22d, -3.33d, 4.44d, -5.55d, 6.66d};
    FILE *stream_1;
    FILE *stream_2;
    stream_1=fopen("file_1.dat", "wb+");
    num1=vargf(stream_1,fmt_1,arr_1[0],
               arr_1[1],
               arr_1[2]);

    if (num1<0)
    {
        printf("An error occurred when calling function vargf first time\n");
    }
    else
    {
        printf("Number of char. printed when vargf is called first time is:%d\n",
               num1);
    }
    stream_2=fopen("file_2.dat", "wb+");

    num2=vargf(stream_2,fmt_2,arr_2[0],
               arr_2[1],
               arr_2[2],
               arr_2[3],
               arr_2[4],
               arr_2[5]);
}

```

Figure 169. ILE C Source to Use the va_arg Macro with a Packed Decimal Data Type (Part 1 of 2)

```

if (num2<0)
{
    printf("An error occurred when calling function vargf second time\n");
}
else
{
    printf("Number of char. printed when vargf is called a second time is:%d\n",
num2);
}
fclose(stream_1);
fclose(stream_2);
}
int vargf(FILE *str, char *fmt,...)
{
    int result;
    va_list arg_ptr;
    va_start(arg_ptr, fmt);
    result = vfprintf(str, fmt, arg_ptr);
    va_end(arg_ptr);
    return result;
}

```

Figure 169. ILE C Source to Use the va_arg Macro with a Packed Decimal Data Type (Part 2 of 2)

The output is as follows:

```

Number of char. printed when vargf is called first time is: 13
Number of char. printed when vargf is called second time is : 27
Press ENTER to end terminal session.

```

The following example shows you how to write packed decimal constants to a file, and how to scan them back. In addition, the example shows you how to pass a packed decimal array to a function.

```

/* This program shows how to write packed decimal constants to a file */
/* and scan them back again. Also shows how to pass a packed decimal */
/* array to a function. */
#include <decimal.h>
#include <stdio.h>
#include <stdlib.h>
#define N    3                /* Array size */
                                /* for decimal declaration. */
FILE *stream;                /* File pointer declaration. */
                                /* Declare valid packed decimal */
                                /* array. */
decimal(4,2) arr_1[] = {12.35d, 25.00d, -19.58d};
decimal(4,2) arr_2[N];
void write_num(decimal(4,2) a[N]); /*Declare function to */
                                /*write to a file. */
void read_num(decimal(4,2) b[N]); /*Declare function to */
                                /*read from a file. */

```

Figure 170. ILE C Source to Write Packed Decimal Constants to a File and Scan Them Back (Part 1 of 2)

```

int main(void)
{
    int reposition=0;
                                /* Open the file.          */
    if ((stream = fopen("CURLIB/OUTFILE","w+")) == NULL)
    {
        printf("Can not open file");
        exit(EXIT_FAILURE);
    }
    write_num(arr_1);            /* Call function to write */
                                /* values of packed decimal */
                                /* array to outfile with fprintf*/
                                /* library function.          */
    reposition=fseek(stream, 0L, SEEK_SET);
    if (reposition!=0)
    {
        printf("FSEEK failed to position file pointer\n");
        exit(EXIT_FAILURE);
    }
    read_num(arr_2);            /* Call function to read */
                                /* values of packed decimal */
                                /* array from file using */
                                /* fscanf() function.          */
    fclose(stream);            /* Close the file.      */
}
/* write_num is passed a packed decimal array. These values are */
/* written to a text file with the fprintf library function.    */
/* If the function is successful a 0 is returned, otherwise a    */
/* negative value is returned (indicating an error).            */

void write_num(decimal(4,2) a[N])
{
    int i, j;

    for (i=0;i < N;i++)
    {
        j = fprintf(stream,"%D(4,2)\n",a[i]);
        if (j < 0)
            printf("Number not written to file %D(4,2)\n",a[i]);
    }
}
/* read_num is passed a packed decimal array. The values are */
/* read from a text file with the fscanf library function.    */
/* If the function is successful a 0 is returned, otherwise a    */
/* negative value is returned (indicating an error).            */

void read_num(decimal(4,2) b[N])
{
    int i, j;
    for (i=0;i < sizeof(b)/sizeof(b[0]);i++)
    {
        j = fscanf(stream,"%D(4,2)\n",&b[i]);
        if (j < 0)
            printf("Error when reading from file\n");
    }
    printf("b[0]=%D(4,2)\nb[1]=%D(4,2)\n\n"
           b[2]=%D(4,2)\n", b[0], b[1], b[2]);
}

```

Figure 170. ILE C Source to Write Packed Decimal Constants to a File and Scan Them Back (Part 2 of 2)

The output is as follows:

```
b[0]=12.35
b[1]=25.00
b[2]=-19.58
Press ENTER to end terminal session.
```

The following example shows how to use the %D(*,*) specifier with the printf() function. If n and p of the variable to be printed do not match with the n and p in the conversion specifier %D(n,p), the behavior is undefined. Use the unary operators digitsof (expression) and precisionof (expression) in the argument list to replace the * in D(*,*) whenever the size of the resulting type of a packed decimal expression is not known.

```
#include <decimal.h>
#include <stdio.h>
int main(void)
{
    decimal(6,2) op_1=1234.12d;
    decimal(10,2) op_2=-12345678.12d;
    printf("op_1 = %*.*D(*,*)\n", 6, 2, digitsof(op_1),
        precisionof(op_1), op_1);
    printf("op_2 = %*.*D(*,*)\n", 10, 2, digitsof(op_2),
        precisionof(op_2), op_2);
}
```

Figure 171. ILE C Source to Print Packed Decimal Constants

The output is as follows:

```
op_1 = 1234.12
op_2 = -12345678.12
Press ENTER to end terminal session.
```

Understanding Packed Decimal Data Type Errors

All the warning and messages for packed decimal data errors are issued by the ILE C compiler during compilation, unless explicitly stated as occurring at run time. If you receive a warning during compilation for overflow, an exception may be generated at run time; SIGFPE is raised. Run-time errors occur during the following packed decimal operations: assignment, casting, initialization, arithmetic operations, and function calls. Some of the overflow situations are signaled during compilation through a warning; loss of digits may occur.

Example

The following example shows all the warnings and error conditions that are issued by the ILE C compiler for packed decimal expressions.

```

#include <decimal.h>
static decimal(5,2) s1 = 12345678.0d; /* No warning or error for */
/* assignment, static or */
/* external initialization.*/

static decimal(10,2) s2 = 1234567891234567891234567.12345d
+ 12345.1234567891d;
/* s2 = (31,5) + (15,10) */
/* Generates an error for */
/* expression, static or */
/* external initialization.*/

static decimal(10,2) s3 = 123456789123456.12345d * 1234567891234.12d;
/* s3 = (20,5) * (15,2) */
/* Generates an error for */
/* multiplication in */
/* static initialization. */

static decimal(10,2) s4 = 12345678912345678912d /
12345.123456789123456d;
/* s4 = (20,0) / (20,15) */
/* Generate an error for */
/* division in static */
/* initialization. */

int main(void)
{
    decimal(5,2) a1 = 12345678.0d; /* No warning or error for */
/* assignment, automatic */
/* initialization, when */
/* the size of the target */
/* is smaller than the size*/
/* of the variable or */
/* packed decimal constant */
/* being assigned. */

    decimal(10,2) a2, a3, a4;
    decimal(5,2) a5 = (decimal(5,2)) 123456.78d;
/* No error or warning */
/* at compilation. */

    a2 = 1234567891234567891234567.12345d + 12345.1234567891d;
/* a2 = (31,5) + (15,10) */
/* Generates an error for */
/* an expression requiring */
/* decimal point alignment.*/

    a3 = 123456789123456.12345d * 1234567891234.12d;
/* a3 = (20,5) * (15,2) */
/* Generate a warning for */
/* multiplication in this */
/* expression. */
/* Note: Need (35,7) but */
/* use (31,2), for example,*/
/* keep the integral part. */
/* Run-time errors are */
/* generated. */

```

Figure 172. Packed Decimal Warnings and Error Conditions (Part 1 of 2)

```

a4 = 12345678912345678912d / 12345.123456789123456d;
/* a4 = (20,0) / (20,15) */
/* Generates an error */
/* during compilation. */
/* Note: Need 35 digits to */
/* calculate integral part */
/* and the result becomes */
/* (31,0). */
}

```

Figure 172. Packed Decimal Warnings and Error Conditions (Part 2 of 2)

Note:

- For assignments to a smaller target field than can hold the packed decimal number, there is no compile-time warning or error issued for static, external or automatic initialization.
- For expressions requiring decimal point alignment, namely addition, subtraction or comparison, there is a compile-time error issued for static, external, and automatic initialization if during alignment, the maximum number of allowed digits is exceeded.
- For multiplication, if the evaluation of the expression results in a value larger than the maximum number of allowed digits:
 - A compile-time error is issued for static or external initialization; no module is created.
 - A compile-time warning is issued if the expression is within a function; a run-time exception is generated.
 - Truncation occurs on the fractional part, preserving as much of the integral part as possible.
- For division, a compile-time error is generated when $((n_1 - p_1) + p_2) > 31$.

The *ILE C for AS/400 Language Reference* contains information on the multiplication and division operators for packed decimals.

Examples

You can use the `#pragma nosigtrunc` directive to suppress a run-time exception that occurs as a result of overflow. The *ILE C for AS/400 Language Reference* contains information on the `#pragma nosigtrunc` directive.

The following example shows how to suppress the run-time exception created when a packed decimal variable overflows on assignment, in a function call, and in an arithmetic operation.

```

/* This program shows how to suppress a run-time exception when a      */
/* packed decimal variable overflows on assignment, in a function call */
/* and in an arithmetic operation.                                     */
#include <decimal.h>
#pragma nosigtrunc                                                     /* The directive turns off */
                                                                    /* SIGFPE which is raised */
                                                                    /* in the following overflow */
                                                                    /* situations; no exception */
                                                                    /* occurs at run time.     */

void f(decimal(4,2) a)
{
}

int main(void)
{
    decimal(8,4) arg=1234.1234d;
    decimal(5,2) op_1=1234567.1234567d; /* Overflow in initialization.*/
    decimal(2) op_2;
    decimal(20,5) op_3=12.34d;
    decimal(15,2) op_4=1234567890.12d;
    decimal(6,2) op_5=1234.12d, cast;
    decimal(31,2) res;
    cast=(decimal(2))op_5; /* Overflow in casting. */
    op_2=arg; /* Overflow in assignment. */
    f(arg); /* Overflow in function call. */
    res=op_3*op_4; /* Overflow in arithmetic */
                /* operation. */
}

```

Figure 173. ILE C Source to Suppress a Run-Time Exception

No run-time exception is logged in the job log.

Chapter 16. Calling Conventions

You can call OPM, EPM or ILE programs using dynamic program calls. A dynamic program call is a call made to a program object (*PGM). Unlike OPM and EPM programs, ILE programs are not limited to using dynamic program calls. ILE programs can use static procedure calls or procedure pointer calls to call other procedures.

This section includes:

- Program and Procedure Calls
- Calling a Program Using a Linkage Specification
- Passing Parameters
- Changing the Names of Programs and Procedures
- Creating C++ Classes for Use in ILE
- Calling OPM Programs
- Calling ILE Programs
- Calling an ILE C++ Program
- Calling an EPM C Program
- Calling ILE-Bindable APIs
- Passing Operational Descriptors

Note: The terms *parameter* and *argument* are used interchangeably.

Program and Procedure Calls

C and C++ are programming languages supported in the Integrated Language Environment (ILE). In ILE it is possible to call either a program (*PGM) or an ILE procedure. The calling program must identify whether the target of the call statement is a program or an ILE procedure. Different C/C++ calling conventions exist for programs and for ILE procedures.

Program processing within ILE occurs at the procedure level. ILE programs consist of one or more modules which in turn consist of one or more procedures. C and C++ modules may contain only one `main()` procedure, but can contain many other procedures (functions). Other ILE languages, however, may allow only one procedure. A dynamic program call is a special form of procedure call; it is a call to the program entry procedure. A program entry procedure is the procedure designated at program creation time to receive control when a program is called. This is the same as calling another program's `main()` function.

Calling Programs

When executing dynamic calls, the called program's name is resolved to an address at run time, just before the calling program passes control to the called program for the first time program calls are dynamic calls.

Calls to an ILE program, an EPM program, or an OPM program are dynamic program calls. A call to a non-bindable API is a dynamic program call.

When an ILE program is called, the program entry procedure receives the program parameters and is given initial control for the program. All procedures within the program become available for procedure calls.

Calling Conventions for Dynamic Program Calls

A **dynamic program call** is a call that is made to a program object (*PGM). A call to an ILE C/C++ program, an EPM program, or an OPM program are all examples of dynamic program calls. A call to a non-bindable API is also an example of a dynamic program call.

You need to know the conventions to call programs. Table 15 shows program call conventions.

Table 15. Program Calling Conventions

Action	Program Call Convention
ILE C calling *PGM where *PGM is <ul style="list-style-type: none"> • ILE C • OPM COBOL for iSeries • OPM RPG for iSeries • OPM CL • OPM BASIC • OPM PL/1 • EPM C • EPM PASCAL • EPM FORTRAN/400® • ILE COBOL for iSeries • ILE RPG for iSeries • ILE CL • C++ 	<pre>#pragma linkage (PGMNAME, OS) For example, #pragma linkage (PGMNAME, OS) void PGMNAME(void); /* Other code */ /* Dynamic call to program PGMNAME */ PGMNAME();</pre>
ILE C calling an EPM entry point	<pre>#pragma linkage (QPXXCALL, OS) For example, #include <xxenv.h> /* The xxenv.h header file holds */ /* the prototype for QPXXCALL */ /* The #pragma linkage (QPXXCALL, OS) */ /* is in this header file. */ /* Other code. */ /* Dynamic call to program QPXXCALL. */ /* Dynamic call to EPM entry point using QPXXCALL: */ /* the name of the entry point is entname, envid */ /* names the user-controlled environment, the */ /* program and library name is given by envpgm, */ /* parm1 and parm2 are arguments passed to entname. */ QPXXCALL(entname, envid, &envpgm, parm1, parm1);</pre>

If you have an ILE C/C++ program calling a program (*PGM) use the #pragma linkage (PGMNAME, OS) directive in your ILE C/C++ source to tell the compiler that PGMNAME is an external program, not a bound Integrated Language Environment procedure.

An ILE C/C++ program passes arguments on a program call with the following conventions:

- If you use the `#pragma linkage (PGMNAME, OS)` directive, all arguments (except pointers and arrays) are copied to temporary variables by the compiler. Pointers to the temporary variables are passed to the called program. Non-pointer arguments are passed by value-reference. Value reference (sometimes referred to as by value, indirectly) refers to the parameter passing mechanism where:
 - A non-pointer value is copied to a temporary variable, and the address of the temporary variable is passed on. The changes that are made to the variable in the called program are not reflected in the calling program.
 - If the argument you are passing is an array name or a pointer, then the argument is passed directly, and a temporary variable is not created. This means that the data that is referred to by the array or pointer can be changed by the called program.
- If you want to pass arguments by reference, you must use the address of (`&`) operator.
- The program name that the ILE C/C++ program calls must be in uppercase. You can use the `#pragma map` directive to map an internal identifier longer than 10 characters to an OS/400 compliant object name (10 characters or less) in your program.
- The return code for the program call can be retrieved by declaring the program to return an integer. For example:

```
#pragma linkage(PGMNAME, OS)
int PGMNAME(void);
```

The value that is returned on the call is the return code for the program call. If the program being called is an Integrated Language Environment program, this return code can also be accessed using the `_LANGUAGE_RETURN_CODE` macro defined in the header file `<milib.h>`. If the program being called is an EPM or OPM program, this return code can be accessed using the Retrieve Job Attributes (RTVJOBA) command.

If you have an ILE C program calling an EPM default entry point, then use the `#pragma linkage (PGMNAME, OS)` directive in your ILE C source to tell the compiler that PGMNAME is an external program, not a bound Integrated Language Environment procedure. QPXXCALL can also be used to call EPM default entry points.

If you have an ILE C program calling an EPM non-default entry point, you must use the EPM API QPXXCALL. Since QPXXCALL is an OPM program, you must use the `#pragma linkage (QPXXCALL, OS)` directive in your ILE C source.

When an ILE C program calls an EPM C program and the EPM program explicitly raises a signal (through the `raise()` function), the ILE program does not monitor for this signal as the EPM environment generates a diagnostic message as a result of the *ESCAPE message generated by the `raise` function. Therefore, the ILE program does not receive any function checks from an EPM program. However, if an ILE C program calls an EPM C program which implicitly raises a signal (as a result of an *ESCAPE message), the *ESCAPE message can be monitored for and handled by the ILE program.

ILE C/C++ programs can be called from any iSeries 400 program. Use the language-specific convention for dynamic program calls to call an ILE C/C++ program.

Example

The following example demonstrates some typical steps in creating an application that uses several languages. The application is a small transaction processing program that takes as input item names, price, and quantity. As output the application displays the total cost of the items that are specified on the screen, and writes an audit trail of transactions to a file.

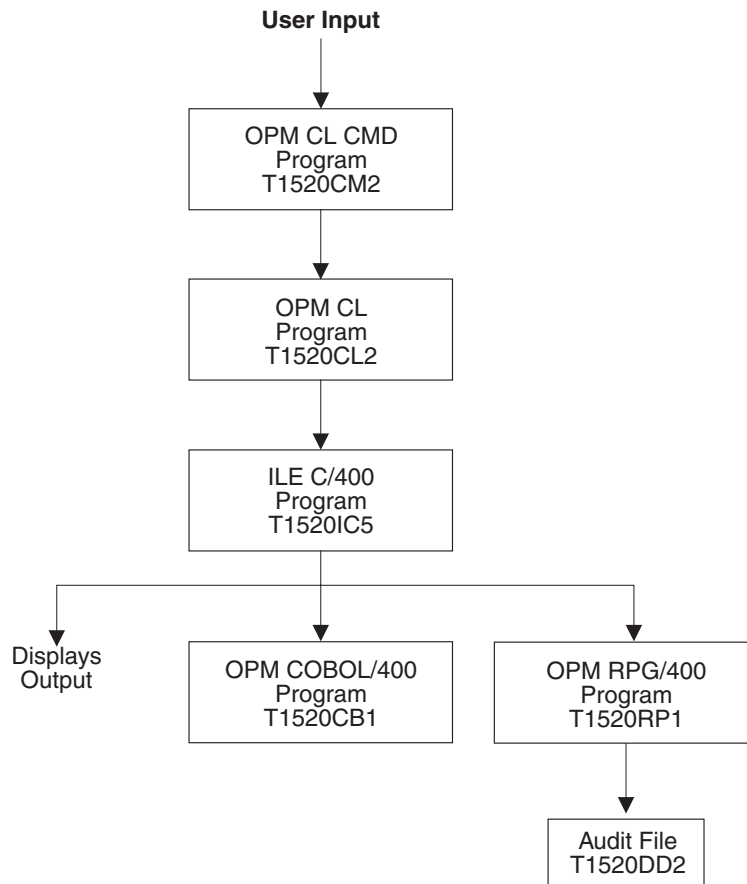


Figure 174. Basic Program Structure

This example consists of:

- A CL command that accepts the user's input and passes it to a CL program.
- A CL program that processes the input and passes it to an ILE C program.
- An ILE C program that calls an OPM COBOL program to process the input, and an OPM RPG program to write the audit trail to an externally described file.
- An OPM COBOL program that completes the calculation and formats the cost.
- An OPM RPG program that updates the audit file with each transaction.

In addition to the source for CMD, CL, ILE C, OPM RPG, and OPM COBOL you need DDS source for the output file.

In the following example the CL, COBOL, and RPG programs are activated within the user default activation groups. A new activation group is started when the CL

programs call the ILE C program because the ILE C program is created with the CRTPGM default of *NEW for the ACTGRP keyword.

Note: When a CRTPGM parameter does not appear in the CRTBNDC command, the CRTPGM parameter default is used.

The following example shows you how to:

- Create a physical file to contain an audit trail for the ILE C program.
- Create a CL program that passes the parameters item name, price, quantity, and user ID to an ILE C program.
- Create a CL command prompt to enter data for item name, price, and quantity. The OPM CL command prompt passes the data to the CL program which in turn calls an ILE C program.
- Create one program with a `main()` function that receives incoming arguments from a CL program, calls an OPM COBOL program to complete the tax calculation and format the total cost. It calls an OPM RPG program to write audit records.
- Create an OPM COBOL program that completes the tax calculation and formats the total cost.
- Create an OPM RPG program that writes the audit trail for the application.

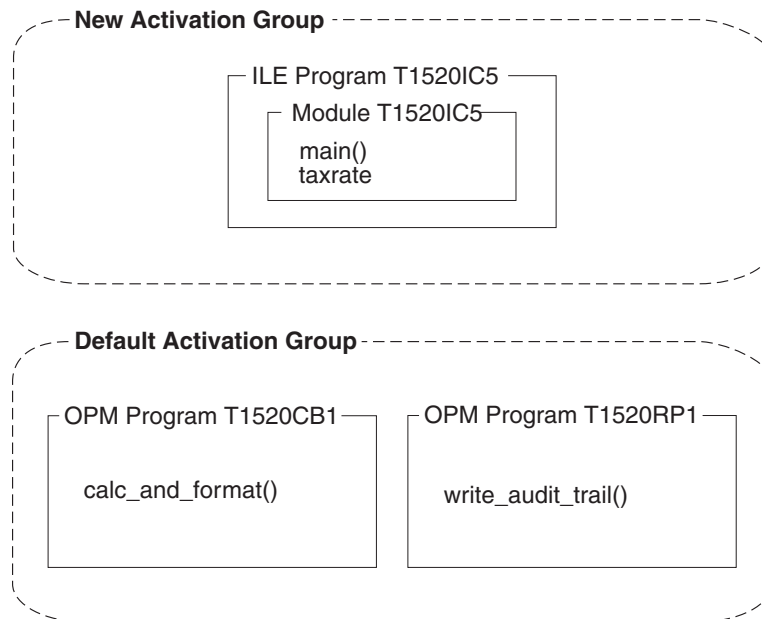


Figure 175. Structure of the Program in ILE C

1. To create a physical file T1520DD2 using the source shown below, type:
`CRTPF FILE(MYLIB/T1520DD2) SRCFILE(QCLE/QADDSSRC) MAXMBRS(*NOMAX)`

This source file contains the audit trail for the ILE C program T1520IC5. The DDS source defines the fields for the audit file.

```

R T1520DD2R
      USER          10          COLHDG('User')
      ITEM           20          COLHDG('Item name')
      PRICE          10S 2       COLHDG('Unit price')
      QTY            4S          COLHDG('Number of items')
      TXRATE         2S 2        COLHDG('Current tax rate')
      TOTAL          21          COLHDG('Total cost')
      DATE           6           COLHDG('Transaction date')
K USER

```

Figure 176. T1520DD2 — DDS Source for an Audit File

2. To create a CL program T1520CL2 using the source shown below, the:

```
CRTCLPGM PGM(MYLIB/T1520CL2) SRCFILE(QCLE/QACLSRC)
```

```

PGM      PARM(&ITEMIN &PRICE &QUANTITY)
        DCL      VAR(&USER) TYPE(*CHAR) LEN(10)
        DCL      VAR(&ITEMIN) TYPE(*CHAR) LEN(20)
        DCL      VAR(&ITEMOUT) TYPE(*CHAR) LEN(21)
        DCL      VAR(&PRICE) TYPE(*DEC) LEN(10 2)
        DCL      VAR(&QUANTITY) TYPE(*DEC) LEN(2 0)
        DCL      VAR(&NULL) TYPE(*CHAR) LEN(1) VALUE(X'00')
        /* ADD NULL TERMINATOR FOR THE ILE C PROGRAM */
        CHGVAR   VAR(&ITEMOUT) VALUE(&ITEMIN *TCAT &NULL)
        /* GET THE USERID FOR THE AUDIT FILE */
        RTVJOBA  USER(&USER)
        /* ENSURE AUDIT RECORDS WRITTEN TO CORRECT AUDIT FILE MEMBER */
        OVRDBF   FILE(T1520DD2) TOFILE(*LIBL/T1520DD2) +
                MBR(T1520DD2) OVRSCOPE(*CALLLVL) SHARE(*NO)
        CALL     PGM(T1520IC5) PARM(&ITEMOUT &PRICE &QUANTITY +
                &USER)
        DLTOVR   FILE(*ALL)
ENDPGM

```

Figure 177. T1520CL2 — CL Source to Pass Variables to an ILE C Program

This program passes the CL variables item name, price, quantity, and user ID by reference to an ILE C program T1520IC5. The Retrieve Job Attributes (RTVJOBA) command obtains the user ID for the audit trail. Arguments are passed by reference. They can be changed by the receiving ILE C program. The variable item_name is null ended in the CL program.

CL variables and numeric constants are not passed to an ILE C program with null-ended strings. Character constants and logical literals are passed as null-ended strings, but are not widened with blanks. Numeric constants such as packed decimals are passed as 15,5 (8 bytes). Floating point constants are passed as double precision floating point values, for example 1.2E+15.

3. To create a CL command prompt T1520CM2 using the source shown below, type:

```
CRTCMD CMD(MYLIB/T1520CM2) PGM(MYLIB/T1520CL2) SRCFILE(QCLE/QACMDSRC)
```

You use this CL command prompt to enter the item name, price, and quantity for the ILE C program T1520IC5.

```

CMD      PROMPT('CALCULATE TOTAL COST')
        PARM      KWD(ITEM) TYPE(*CHAR) LEN(20) RSTD(*NO) +
                   MIN(1) ALWUNPRT(*NO) PROMPT('Item name' 1)
        PARM      KWD(PRICE) TYPE(*DEC) LEN(10 2) RSTD(*NO) +
                   RANGE(0.01 99999999.99) MIN(1) +
                   ALWUNPRT(*YES) PROMPT('Unit price' 2)
        PARM      KWD(QUANTITY) TYPE(*INT2) RSTD(*NO) RANGE(1 +
                   9999) MIN(1) ALWUNPRT(*YES) +
                   PROMPT('Number of items' 3)

```

Figure 178. T1520CM2 — CL Command Source to Received Input Data

4. To create the program T1520IC5 using the source shown below, type:

```

CRTBNDC PGM(MYLIB/T1520IC5) SRCFILE(QCLE/QACSRC) OUTPUT(*PRINT) FLAG(30)
        OPTION(*EXPMAC *SHOWINC *NOLOGMSG) MSGLMT(10) CHECKOUT(*PARM) DBGVIEW(*ALL)

```

```

/* This program is called by a CL program that passes an item      */
/* name, price, quantity and user ID.                               */
/* COBOL for iSeries 400 is called to calculate and format the total cost. */
/* RPG for iSeries 400 is called to write an audit trail.           */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <decimal.h>
/* The #pragma map directive maps a new program name to the existing */
/* program name so that the purpose of the program is clear.         */
#pragma map(calc_and_format,"T1520CB1")
#pragma map(write_audit_trail,"T1520RP1")
/* Tell the compiler that there are dynamic program calls so      */
/* arguments are passed by value-reference.                         */
#pragma linkage(calc_and_format, OS, nowiden)
#pragma linkage(write_audit_trail, OS)
void calc_and_format(decimal (10,2),
                    short int,
                    decimal(2,2),
                    char[],
                    char *);
void write_audit_trail(char[],
                    char[],
                    decimal(10,2),
                    short int,
                    decimal(2,2),
                    char[]);
int main(int argc, char *argv[])
{
/* Incoming arguments from a CL program have been verified by      */
/* the *CMD and null end within the CL program.                    */
/* Incoming arguments are passed by reference from a CL program.    */
char      *user_id;
char      *item_name;
short int  quantity;
decimal (10, 2) price;
decimal (2,2) taxrate = .15D;
char      formatted_cost[22];

```

Figure 179. T1520IC5 — ILE C Source to Call COBOL AND RPG (Part 1 of 2)

```

/* Remove null end for RPG for iSeries 400 program. Item name is null */
/* ended for C. */

char      rpg_item_name[20];
char      null_formatted_cost[22];
char      success_flag = 'N';
int        i;

/* Incoming arguments are all pointers. */
item_name = argv[1];
price      = *((decimal (10, 2) *) argv[2]);
quantity   = *((short *) argv[3]);
user_id    = argv[4];

/* Call the COBOL for iSeries 400 program to do the calculation, and return a */
/* Y/N flag, and a formatted result. */

    calc_and_format(price,
                    quantity,
                    taxrate,
                    formatted_cost,
                    &success_flag);
    memcpy(null_formatted_cost,formatted_cost,sizeof(formatted_cost));

/* Null end the result. */
formatted_cost[21] = '\0';
if (success_flag == 'Y')
{
    for (i=0; i<20; i++)
    {

/* Remove null end for the RPG for iSeries 400 program. */
        if (*(item_name+i) == '\0')
        {
            rpg_item_name[i] = ' ';
        }
        else
        {
            rpg_item_name[i] = *(item_name+i);
        }
    }

/* Call an RPG program to write audit records. */
    write_audit_trail(user_id,
                    rpg_item_name,
                    price,
                    quantity,
                    taxrate,
                    formatted_cost);

    printf("\n%d %s plus tax = %-s\n", quantity,
                    item_name,
                    null_formatted_cost);
}
else
{
    printf("Calculation failed\n");
}
}

```

Figure 179. T1520IC5 — ILE C Source to Call COBOL AND RPG (Part 2 of 2)

The `main()` function in this program receives incoming arguments from CL program T1520CL2 that are verified by CL command prompt T1520CM2 and null ended within CL program T1520CL2. All incoming arguments are pointers. The `main()` function also calls `calc_and_format()`, which is mapped to a COBOL name. It passes by OS-linkage convention the price, quantity, taxrate, formatted cost, and a success flag.

Because the OPM COBOL program is not expecting widened parameters (the default for ILE C), `nowiden` is used in the `#pragma linkage` directive. The formatted cost and the success flag values are updated in program T1520IC5.

If `calc_and_format()` returns successfully a record is written to the audit trail by `write_audit_trail()` in the OPM RPG program. The `main()` function in this program (T1520IC5) calls `write_audit_trail()` which is mapped to an RPG for iSeries 400 program name. It passes by OS-linkage convention the user ID, item name, price, quantity, taxrate, and formatted cost.

The ILE Compiler by default converts a short integer to an integer unless the `nowiden` parameter is specified on the `#pragma linkage` directive. For example, the short integer in the ILE C program is converted to an integer, and then passed to the OPM RPG program. The RPG program is expecting a 4 byte integer for the quantity variable.

`OUTPUT(*PRINT)` specifies that you want a compiler listing. `OPTION(*EXPMAC *SHOWINC *NOLOGMSG)` specifies that you want to expand include files and macros in a compiler listing, and not log CHECKOUT option messages.

`FLAG(30)` specifies that you want severity level 30 messages to appear in the listing. `MSGLMT(10)` specifies that you want compilation to stop after 10 messages at severity level 30. `CHECKOUT(*PARM)` shows a list of function parameters that are not used. `DBGVIEW(*ALL)` specifies that you want all three views and debug data to debug this program.

5. To create an OPM COBOL program using the source shown below, type:
`CRTCLPGM PGM(MYLIB/T1520CB1) SRCFILE(QCLE/QALBLSRC)`

```
IDENTIFICATION DIVISION.
    PROGRAM-ID. T1520CB1.
    *****
    * parameters:                                     *
    *   incoming:  PRICE, QUANTITY                     *
    *   returns :  TOTAL-COST (PRICE*QUANTITY*1.TAXRATE) *
    *              SUCCESS-FLAG.                       *
    *****
    ENVIRONMENT DIVISION.                                0
    CONFIGURATION SECTION.                              0
        SOURCE-COMPUTER. AS-400.                        0
        OBJECT-COMPUTER. AS-400.                        0
    DATA DIVISION.                                     0
    WORKING-STORAGE SECTION.
        01 WS-TOTAL-COST          PIC S9(13)V99          COMP-3.
        01 WS-TAXRATE             PIC S9V99              COMP-3
```

Figure 180. T1520CB1 — OPM COBOL Source to Calculate Tax and Format Cost (Part 1 of 2)

```

LINKAGE SECTION.
01 LS-PRICE          PIC S9(8)V9(2)      COMP-3.
01 LS-QUANTITY       PIC S9(4)          COMP-4.
01 LS-TAXRATE        PIC S9(9)          COMP-3.
01 LS-TOTAL-COST     PIC $$$,$$$,$$$,$$$,$$.99
                                DISPLAY.

01 LS-OPERATION-SUCCESSFUL PIC X        DISPLAY.
PROCEDURE DIVISION USING LS-PRICE          0
                        LS-QUANTITY
                        LS-TAXRATE
                        LS-TOTAL-COST
                        LS-OPERATION-SUCCESSFUL.

MAINLINE.
  MOVE "Y" TO LS-OPERATION-SUCCESSFUL.
  PERFORM CALCULATE-COST.
  PERFORM FORMAT-COST.
  EXIT PROGRAM.
CALCULATE-COST.
  MOVE LS-TAXRATE TO WS-TAXRATE.
  ADD 1 TO WS-TAXRATE.
  COMPUTE WS-TOTAL-COST ROUNDED = LS-QUANTITY *
                                LS-PRICE *
                                WS-TAXRATE

  ON SIZE ERROR
    MOVE "N" TO LS-OPERATION-SUCCESSFUL
  END-COMPUTE.
FORMAT-COST.
  MOVE WS-TOTAL-COST TO LS-TOTAL-COST.

```

Figure 180. T1520CB1 — OPM COBOL Source to Calculate Tax and Format Cost (Part 2 of 2)

This program receives pointers to the values of the variables price, quantity, and taxrate, and pointers to formatted_cost and success_flag.

The calc_and_format() function in program T1520CB1 calculates and formats the total cost. Parameters are passed from the ILE C program to the OPM COBOL program to do the tax calculation.

The *ILE C for AS/400 Run-Time Library Reference* contains information on how to compile an OPM COBOL program.

6. To create an OPM RPG program using the source shown below, type:
CRTRPGPGM PGM(MYLIB/T1520RP1) SRCFILE(QCLE/QARPGSRC) OPTION(*SOURCE *SECLVL)

```

FT1520DD20  E          DISK          A
F          T1520DD2R          KRENAMEDD2R
I QTYIN      DS
I          B  1  40QTYBIN
C          *ENTRY  PLIST
C          PARM      USER  10
C          PARM      ITEM  20
C          PARM      PRICE 102
C          PARM      QTYIN
C          PARM      TXRATE 22
C          PARM      TOTAL 21
C          EXSR ADDREC
C          SETON          LR
C          ADDREC  BEGSR
C          MOVE LDATE  DATE
C          MOVE QTYBIN QTY
C          WRITEDD2R
C          ENDSR

```

Figure 181. T1520RP1 — OPM RPG Source to Write the Audit Trail

The `write_audit_trail()` function in the program T1520RP1 writes the audit trail for the program.

The *RPG/400 User's Guide* contains information on how to compile an OPM RPG program.

7. To enter data for the program T1520IC5, type:

T1520CM2

and press F4 (Prompt).

Type the following data into T1520CM2:

```

Hammers
1.98
5000
Nails
0.25
2000

```

The output is as follows:

```

5000 HAMMERS plus tax =          $11,385.00
Press ENTER to end terminal session.
>
2000 NAILS plus tax =           $575.00
Press ENTER to end terminal session.

```

The physical file T1520DD2 contains the data as follows:

```

SMITHE  HAMMERS          00000000198500015          $11,385.00072893
SMITHE  NAILS           0000000025200015          $575.00072893

```

This example is an ILE version of the small transaction processing program that is given in the previous example.

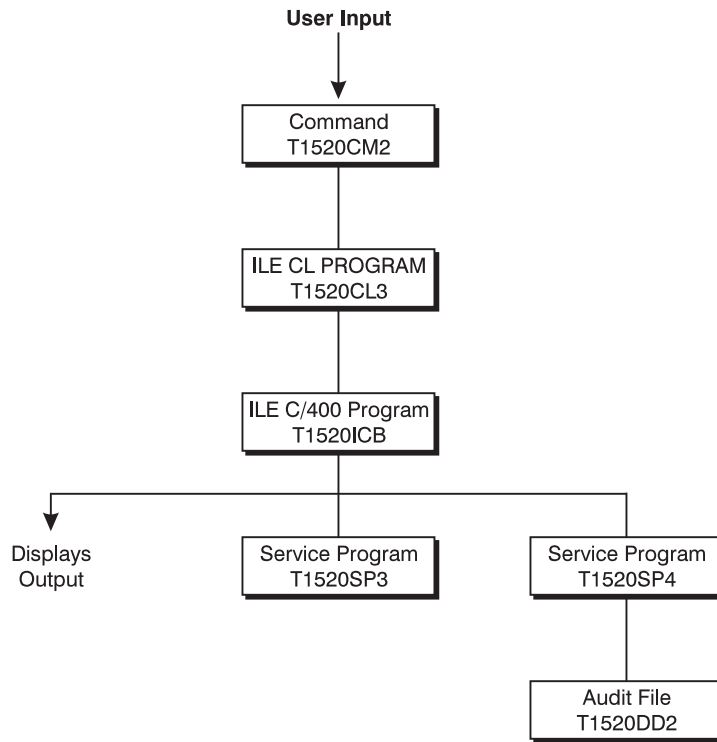


Figure 182. Basic Object Structure

This example consists of:

- A CL command that accepts user's input and passes it to an ILE CL program.
- A ILE CL program that processes the input and passes it to an ILE C program.
- The ILE C program calls Integrated Language Environment procedures to process the input. Output is then written to the user's terminal.
- An Integrated Language Environment COBOL procedure that completes the calculation and formats the cost.
- An Integrated Language Environment RPG procedure that updates the audit file with each transaction.

In addition to the source for CMD, ILE CL, ILE C, Integrated Language Environment RPG and ILE COBOL you need DDS source for the output file which is the same as the previous example.

In the following example the CL and C programs are activated within a new activation group. The Integrated Language Environment CL program is created with the CRTPGM default for the ACTGRP parameter, ACTGRP(*NEW). The ILE C program is created with ACTGRP(*CALLER)

The following example shows you how to:

- Create a physical file to contain an audit trail for the ILE C program.
- Create a Integrated Language Environment CL program that passes the parameters item name, price, quantity, and user ID to an ILE C program.
- Create a CL command prompt to enter data for item name, price, and quantity. The command passes the data to the Integrated Language Environment CL program which in turn calls an ILE C program.

- Create an Integrated Language Environment COBOL module that completes the tax calculation and formats the total cost.
- Create an Integrated Language Environment RPG module that writes the audit trail for the application.
- Create a service program that exports a data item.
- Create a service program that exports an RPG procedure.
- Create one program with a `main()` function that receives incoming arguments from a CL program. The program calls an Integrated Language Environment COBOL procedure to complete the tax calculation and format the total cost, and calls an ILE RPG procedure to write audit records.

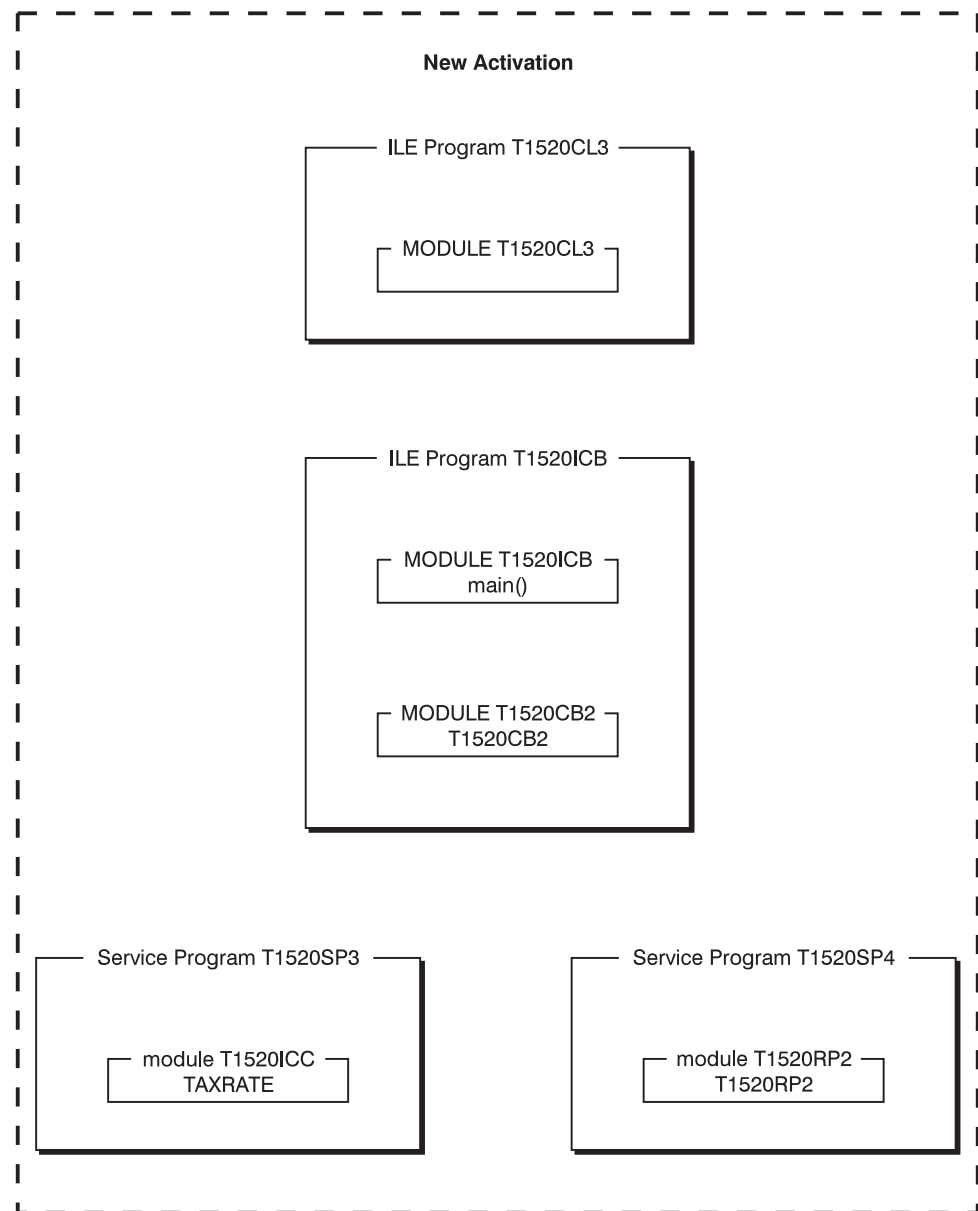


Figure 183. Integrated Language Environment Structure

1. To create a physical file T1520DD2 using the source shown below, type:
`CRTPF FILE(MYLIB/T1520DD2) SRCFILE(QCLE/QADDSSRC) MAXMBRS(*NOMAX)`

```

R T1520DD2R
      USER          10          COLHDG('User')
      ITEM           20          COLHDG('Item name')
      PRICE          10S 2       COLHDG('Unit price')
      QTY            4S          COLHDG('Number of items')
      TXRATE         2S 2       COLHDG('Current tax rate')
      TOTAL          21          COLHDG('Total cost')
      DATE           6          COLHDG('Transaction date')
K USER

```

This file contains the audit trail for the ILE C program T1520ICB.

2. To create a CL program T1520CL3 that the source shown below, type:

```

CRTCLMOD MODULE(MYLIB/T1520CL3) SRCFILE(QCLE/QACLSRC)
CRTPGM PGM(MYLIB/T1520CL3) MODULE(MYLIB/T1520CL3) ACTGRP(*NEW)

```

```

/* ILE version of T1520CL2                                     */
PGM          PARM(&ITEMIN &PRICE &QUANTITY)
DCL          VAR(&USER) TYPE(*CHAR) LEN(10)
DCL          VAR(&ITEMIN) TYPE(*CHAR) LEN(20)
DCL          VAR(&ITEMOUT) TYPE(*CHAR) LEN(21)
DCL          VAR(&PRICE) TYPE(*DEC) LEN(10 2)
DCL          VAR(&QUANTITY) TYPE(*DEC) LEN(2 0)
DCL          VAR(&NULL) TYPE(*CHAR) LEN(1) VALUE(X'00')
/* ADD NULL TERMINATOR FOR THE ILE C PROGRAM                */
CHGVAR      VAR(&ITEMOUT) VALUE(&ITEMIN *TCAT &NULL)
/* GET THE USERID FOR THE AUDIT FILE                        */
RTVJQBA     USER(&USER)
/* ENSURE AUDIT RECORDS WRITTEN TO CORRECT AUDIT FILE MEMBER */
OVRDBF      FILE(T1520DD2) TOFILE(*LIBL/T1520DD2) +
            MBR(T1520DD2) OVRSCOPE(*CALLLVL) SHARE(*NO)
CALL        PGM(T1520ICB) PARM(&ITEMOUT &PRICE &QUANTITY +
            &USER)
DLTOVR      FILE(*ALL)
ENDPGM

```

Figure 184. T1520CL3 — ILE CL Source to Pass Variables to an ILE C Program

This program passes the CL variables item name, price, quantity, and user ID by reference to an ILE C program T1520ICB. The Retrieve Job Attributes (RTVJQBA) command obtains the user ID for the audit trail. Arguments are passed by reference. They can be changed by the receiving ILE C program. The variable item_name is null ended in the CL program.

3. To create a CL command prompt T1520CM2 using the source below, type:

```

CRTCMD CMD(MYLIB/T1520CM2) PGM(MYLIB/T1520CL3) SRCFILE(QCLE/QACMDSRC)

```

```

CMD          PROMPT('CALCULATE TOTAL COST')
          PARM      KWD(ITEM) TYPE(*CHAR) LEN(20) RSTD(*NO) +
                    MIN(1) ALWUNPRT(*NO) PROMPT('Item name' 1)
          PARM      KWD(PRICE) TYPE(*DEC) LEN(10 2) RSTD(*NO) +
                    RANGE(0.01 99999999.99) MIN(1) +
                    ALWUNPRT(*YES) PROMPT('Unit price' 2)
          PARM      KWD(QUANTITY) TYPE(*INT2) RSTD(*NO) RANGE(1 +
                    9999) MIN(1) ALWUNPRT(*YES) +
                    PROMPT('Number of items' 3)

```

You use this CL command to enter the item name, price, and quantity for the ILE C program T1520ICB.

4. To create the module T1520ICB using the source shown below, type:

```
CRTCMOD MODULE(MYLIB/T1520ICB) SRCFILE(QCLE/QACSRC)
```

```
/* This program demonstrates the interlanguage call capability      */
/* of an ILE C program. This program is called by a CL              */
/* program that passes an item name, price, quantity and user ID.  */
/* A COBOL procedure is called to calculate and format total cost.*/
/* An RPG procedure is called to write an audit trail.              */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <decimal.h>
/* The #pragma map directive maps a function name to the bound      */
/* procedure name so that the purpose of the procedure is clear.    */
#pragma map(calc_and_format,"T1520CB2")
#pragma map(write_audit_trail,"T1520RP2")
/* Tell the compiler that there are bound procedure calls and      */
/* arguments are to be passed by value-reference.                  */
#pragma argument(calc_and_format, 0S, nowiden)
#pragma argument(write_audit_trail, 0S)
void calc_and_format(decimal (10,2),
                    short int,
                    char[],
                    char *);
void write_audit_trail(char[],
                    char[],
                    decimal(10,2),
                    short int,
                    char[]);
extern decimal (2,2) TAXRATE; /* TAXRATE is in *SRVPGM T1520SP3 */
int main(int argc, char *argv[])
{
/* Incoming arguments from a CL program have been verified by      */
/* the *CMD and null ended within the CL program.                  */
/* Incoming arguments are passed by reference from a CL program.    */
char      *user_id;
char      *item_name;
short int  quantity;
decimal (10, 2) price;
char      formatted_cost[22];
/* Remove null terminator for RPG for iSeries 400 program. Item name is null */
/* ended for C.                                                      */
char      rpg_item_name[20];
char      null_formatted_cost[22];
char      success_flag = 'N';
int        i;
/* Incoming arguments are all pointers.                              */
item_name = argv[1];
price     = *((decimal (10, 2) *) argv[2]);
quantity  = *((short *) argv[3]);
user_id   = argv[4];
```

Figure 185. T1520ICB — ILE C Source to Call COBOL and RPG Procedures (Part 1 of 2)

```

/* Call the COBOL program to do the calculation, and return a      */
/* Y/N flag, and a formatted result.                                */
calc_and_format(price,
                 quantity,
                 formatted_cost,
                 &success_flag);

memcpy(null_formatted_cost,formatted_cost,sizeof(formatted_cost));

/* Null end the result.                                           */

formatted_cost[21] = '\0';
if (success_flag == 'Y')
{
    for (i=0; i<20; i++)
    {
/* Remove the null end for the RPG for iSeries 400 program.      */
        if (*(item_name+i) == '\0')
        {
            rpg_item_name[i] = ' ';
        }
        else
        {
            rpg_item_name[i] = *(item_name+i);
        }
    }
}

/* Call an RPG program to write audit records.                    */

write_audit_trail(user_id,
                  rpg_item_name,
                  price,
                  quantity,
                  formatted_cost);

printf("\n%d %s plus tax = %-s\n", quantity,
                                             item_name,

}
else
{
    printf("Calculation failed\n");
}
}

```

Figure 185. T1520ICB — ILE C Source to Call COBOL and RPG Procedures (Part 2 of 2)

The `main()` function in this module receives the incoming arguments from the Integrated Language Environment CL program T1520CL3 that are verified by the CL command T1520CM2 and null ended within the CL program T1520CL3. All the incoming arguments are pointers.

The `main()` function in this program calls `calc_and_format()` which is mapped to a Integrated Language Environment COBOL procedure name. It passes by OS-linkage convention the price, quantity, formatted cost, and a success flag. The Integrated Language Environment OPM COBOL procedure is not expecting widened parameters, the default for ILE C. This is why `nowiden` is used in the `#pragma` argument directive. The formatted cost and the success flag values are updated in the procedure T1520CB2.

If `calc_and_format()` return successfully the `main()` function in this program (T1520ICB) calls `write_audit_trail()` which is mapped to an Integrated Language Environment RPG procedure name. It passes by OS-linkage convention (also called by value-reference) the user ID, item name, price, quantity, and formatted cost. The ILE C compiler by default converts a short integer to an integer unless the `nowiden` parameter is specified on the `#pragma` argument directive. For example, the short integer in the ILE C program is converted to an integer, and then passed to the ILE RPG procedure. The RPG procedure is expecting a 4 byte integer for the quantity variable.

5. To create module T1520ICC using the source shown below, type:

```
CRTCMOD MODULE(MYLIB/T1520ICC) SRCFILE(QCLE/QACSRC)
```

```
TAXRATE is exported from this module to Integrated Language Environment
/* Export the tax rate data.                                     */
#include <decimal.h>
const decimal (2,2) TAXRATE = .15D;
```

C, COBOL, and RPG procedures.

Note: Weak definitions (EXTERNALs from COBOL) cannot be exported out of a service program to a strong definition language like C. C can export to COBOL, hence the choice of language for TAXRATE.

6. To create an Integrated Language Environment COBOL procedure using the source shown below, type:

```
CRTCBMOD MODULE(MYLIB/T1520CB2) SRCFILE(QCLE/QALBLSRC)
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID. T1520CB2 INITIAL.
*****
* parameters:                                     *
*   incoming:  PRICE, QUANTITY                     *
*   returns :  TOTAL-COST (PRICE*QUANTITY*1.TAXRATE) *
*             SUCCESS-FLAG.                         *
*   TAXRATE :  An imported value.                   *
*****
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. AS-400.
OBJECT-COMPUTER. AS-400.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-TOTAL-COST          PIC S9(13)V99          COMP-3.
01 WS-TAXRATE             PIC S9V99              COMP-3
                                VALUE 1.
01 TAXRATE                EXTERNAL PIC SV99      COMP-3.
LINKAGE SECTION.
01 LS-PRICE               PIC S9(8)V9(2)         COMP-3.
01 LS-QUANTITY            PIC S9(4)              COMP-4.
01 LS-TOTAL-COST          PIC $$$,$$$,$$$,$$$,$$.99
                                DISPLAY.
01 LS-OPERATION-SUCCESSFUL PIC X                DISPLAY.
```

Figure 186. T1520CB2 — ILE COBOL Source to Calculate Tax and Format Cost (Part 1 of 2)

```

PROCEDURE DIVISION USING LS-PRICE
                        LS-QUANTITY
                        LS-TOTAL-COST
                        LS-OPERATION-SUCCESSFUL.

MAINLINE.
  MOVE "Y" TO LS-OPERATION-SUCCESSFUL.
  PERFORM CALCULATE-COST.
  PERFORM FORMAT-COST.
  EXIT PROGRAM.
CALCULATE-COST.
  ADD TAXRATE TO WS-TAXRATE.
  COMPUTE WS-TOTAL-COST ROUNDED = LS-QUANTITY *
                                LS-PRICE *
                                WS-TAXRATE

  ON SIZE ERROR
    MOVE "N" TO LS-OPERATION-SUCCESSFUL
  END-COMPUTE.
FORMAT-COST.
  MOVE WS-TOTAL-COST TO LS-TOTAL-COST.

```

Figure 186. T1520CB2 — ILE COBOL Source to Calculate Tax and Format Cost (Part 2 of 2)

This program receives pointers to the values of the variables price and quantity, and pointers to formatted_cost and success_flag.

The calc_and_format() function is procedure T1520CB2. It calculates and formats the total cost.

The *ILE COBOL Programmer's Guide* contains information on how compile an Integrated Language Environment COBOL source program.

7. To create an Integrated Language Environment RPG procedure using the source shown below, type:

```
CRTRPGMOD MODULE(MYLIB/T1520RP2) SRCFILE(QCLE/QARPGSRC)
```

```

FT1520DD2  O A E          DISK
D TAXRATE          S          3P 2 IMPORT
D QTYIN            DS
D QTYBIN           1        4B 0
C   *ENTRY         PLIST
C                   PARM          USER          10
C                   PARM          ITEM          20
C                   PARM          PRICE         10 2
C                   PARM          QTYIN
C                   PARM          TOTAL          21
C                   EXSR          ADDREC
C                   SETON
C                   BEGSR
C   ADDREC         MOVE          UDATE          DATE
C                   MOVE          QTYBIN         QTY
C                   MOVE          TAXRATE        TXRATE
C                   WRITE          T1520DD2R
C                   ENDSR

```

Figure 187. T1520RP2 — ILE RPG Source to Write the Audit Trail

The write_audit_trail() function is the procedure T1520RP2. It writes the audit trail for the program.

The *ILE RPG Programmer's Guide* contains information on how to compile an Integrated Language Environment RPG source program.

8. To create the service program T1520SP3 from the module T1520ICC, type:
CRTSRVPGM SRVPGM(MYLIB/T1520SP3) MODULE(MYLIB/T1520ICC) +
EXPORT(*SRCFILE) SRCFILE(QCLE/QASRVSRC)

The T1520SP3 service program exports taxrate. The export list is specified in T1520SP3 in QASRVSRC.

9. To create the service program T1520SP4 from the module T1520RP2, type:
CRTSRVPGM SRVPGM(MYLIB/T1520SP4) MODULE(MYLIB/T1520RP2) +
EXPORT(*SRCFILE) SRCFILE(QCLE/QASRVSRC)

The T1520SP4 service program exports procedure T1520RP2. The export list is specified in T1520SP4 in QASRVSRC.

10. To create the program T1520ICB type:
CRTPGM PGM(MYLIB/T1520ICB) MODULE(MYLIB/T1520ICB MYLIB/T1520CB2) +
BNDSRVPGM(MYLIB/T1520SP3 MYLIB/T1520SP4) ACTGRP(*CALLER)

T1520ICB is considered the applications main program. It will run in the new activation group that was created when T1520CL3 was called.

11. To enter data for the program T1520ICB, type T1520CM2 and press F4 (Prompt):
Type the following data into T1520CM2:

```
Hammers
1.98
5000
Nails
0.25
2000
```

The output is as follows:

```
5000 HAMMERS plus tax =          $11,385.00
Press ENTER to end terminal session.
>
2000 NAILS plus tax =           $575.00
Press ENTER to end terminal session.
```

The physical file T1520DD2 contains the data as follows:

SMITHE	HAMMERS	0000000198500015	\$11,385.00072893
SMITHE	NAILS	0000000025200015	\$575.00072893

Example

The following example shows how to retrieve a return value from `main()`. A CL command called `SQUARE` calls an ILE C program `SQITF`. The program `SQITF` calls another ILE C program called `SQ`. The program `SQ` returns a value to program `SQITF`.

Note: Returning an integer value from an ILE C program may affect performance.

1. To create a CL command prompt `SQUARE` using the source shown below, type:
CRTCMD CMD(MYLIB/SQUARE) PGM(MYLIB/SQITF) SRCFILE(MYLIB/QCMDSRC)

```

CMD      PROMPT('CALCULATE THE SQUARE')
        PARM      KWD(VALUE) TYPE(*INT4) RSTD(*NO) RANGE(1 +
                    9999) MIN(1) ALWUNPRT(*YES) PROMPT('Value' 1)

```

Figure 188. SQUARE — CL Command Source to Receive Input Data

You use the CL command prompt SQUARE to enter the item name, price, and quantity for the ILE C program SQITF.

2. To create a program SQIFT using the source shown below, type:

```

CRTBNDC PGM(MYLIB/SQIFT) SRCFILE(MYLIB/QCSRC)

```

```

/* This program SQITF is called by the command SQUARE. This      */
/* program then calls another ILE C program SQ to perform        */
/* calculations and return a value.                               */
#include <stdio.h>
#include <decimal.h>
#pragma linkage(SQ, OS)      /* Tell compiler this is external call, */
                             /* do not pass by value.                */
int SQ(int);
int main(int argc, char *argv[])
{
    int  *x;
    int  result;
    x = (int *) argv[1];
    result = SQ(*x);
    /* Note that although the argument is passed by value, the compiler */
    /* copies the argument to a temporary variable, and the pointer to */
    /* the temporary variable is passed to the called program SQ.      */
    printf("The SQUARE of %d is %d\n", *x, result);
}

```

Figure 189. SQITF — ILE C Source to Pass an Argument by Value

3. To create the program SQ using the source shown below, type:

```

CRTBNDC PGM(MYLIB/SQ) SRCFILE(MYLIB/QCSRC) OUTPUT(*PRINT)

```

```

/* This program is called by another ILE C program called SQITF. */
/* It performs the square calculations and returns a value to SQITF. */
#include <stdio.h>
#include <decimal.h>
int main(int argc, char *argv[])
{
    int  *vin;
    int  vout;
    vin = (int *) argv[1];
    vout = (*vin)*(*vin);
    return(vout);
}

```

Figure 190. SQ — ILE C Source to Perform Calculations and Return a Value

The program SQ calculates an integer value and returns the value to the calling program SQITF.

4. To enter data for the program SQITF, type:

```

SQUARE

```

and press F4 (Prompt).

5. Type 10, and press Enter. The output is as follows:

```
The SQUARE of 10 is 100
Press ENTER to end terminal session.
```

Calling Procedures for ILE C

ILE C programs are called by dynamic program calls, but the procedures within an activated ILE C program can be accessed using static procedure calls or procedure pointer calls. ILE C programs that have not been activated yet must be called by a dynamic program call.

In contrast to static procedure calls, which are resolved and bound at compile time, symbols for dynamic program calls are resolved to addresses when the call is performed. As a result, a dynamic program call uses more system resources at run time (specifically at program activation) than a static procedure call.

Note: EPM C and Pascal procedures or functions cannot call ILE C procedures. OPM programs cannot call any ILE procedures (including ILE C procedures).

A **static procedure call** can call a procedure within the same module, a procedure in a separate module within the same ILE C program or service program, or a procedure in a separate ILE C service program.

The term procedure in ILE is equivalent to the term function in ILE C.

ILE C allows arguments to be passed between procedures that are written in different Integrated Language Environment HLLs. The calling function must make sure that the arguments are the size and type that are expected by the called function.

ILE C provides a `#pragma` argument directive to simplify calls to bound procedures in languages such as ILE COBOL and ILE RPG. The `#pragma` argument directive allows arguments to be passed by mechanisms other than the standard C 'by value, directly' mechanism.

By default, ILE C procedures pass and accept arguments **by value**, widening integers and floating point values. "By value" means that the value of the data object is placed directly into the argument list.

Integrated Language Environment RPG passes arguments **by reference** and accepts arguments by reference. "By reference" means a pointer to the data object is placed into the argument list. Changes that are made by the called procedure to the argument are reflected in the calling procedure. Additionally, ILE RPG can pass arguments by value.

Integrated Language Environment COBOL passes arguments by reference, and pass **by content** (by value, indirectly). "By content" means that the value of the data object is copied to a temporary location. The address of the copy, a pointer, is placed into the argument list. Additionally, ILE COBOL can pass arguments by value.

Table 16 on page 346 shows the default argument that passes methods on procedure calls.

Table 16. Argument Passing for Integrated Language Environment Procedures

Integrated Language Environment HLL	Pass Argument	Receive Argument
ILE C default	By value	By value
ILE C with #pragma argument OS directive	Use OS-linkage convention for Integrated Language Environment bound procedure calls only.	By value
ILE C with #pragma linkage OS directive	Use OS-linkage when calling external programs.	By reference
Integrated Language Environment COBOL default	By reference	By reference
Integrated Language Environment CL	By reference	By reference
Integrated Language Environment RPG default	By reference	By reference

Procedure pointer calls provide a way to call a procedure dynamically. For example, by manipulating arrays or tables of procedure names or addresses, you can dynamically route a procedure call to different procedures.

Operational descriptors provide descriptive information to the called procedure in cases where the called procedure cannot precisely anticipate the form of the argument, for example, different types of strings. The additional information allows the procedure to properly interpret the string. You should only use operational descriptors when they are expected by the called procedure, usually an Integrated Language Environment bindable API.

ILE C provides a #pragma descriptor directive to identify functions whose arguments have operational descriptors. You can retrieve the information from an operational descriptor using the Integrated Language Environment bindable APIs Retrieve Operational Descriptor Information (CEEDOD) and Get Descriptive Information About a String Argument (CEESGI). The ILE C compiler supports operational descriptors for string arguments.

Example

The following example shows you how to call a Dynamic Screen Manager (DSM) ILE bindable API to display a DSM session. This DSM session echoes back data that you enter during the DSM session.

1. To create module T1520API using the source shown below, type:
CRTCMOD MODULE(MYLIB/T1520API) SRCFILE(QCLE/QACSRC) OUTPUT(*PRINT)

```
/* This program uses Dynamic Screen Manager API calls to */
/* create a window and echo whatever is entered. This is an */
/* example of bound API calls. Note the use of #pragma argument */
/* in the <qsnsess.h> header file. OS, nowiden ensures that a pointer */
/* to an unwidened copy of the argument is passed to the API. */
```

Figure 191. T1520API — ILE C Source to Call an ILE C Procedure (Part 1 of 3)

```

/*                                                                    */
/* Use BNDDIR(QSNAPI) on the CRTPGM command to build this            */
/* example.                                                            */
#include <stddef.h>
#include <string.h>
#include <stdio.h>
#include "QSYSINC/H/QSNAPI"

/* QSNSESS nests QSNWIN and QSNLL include files. To get these 3      */
/* include files, do the following:                                    */
/* 1) If you do not have a SRCPF called H in your Library (MYLIB),    */
/*    create one.                                                      */
/* 2) Copy QUSRT00L/QATTSYSC/OPSN3API to MYLIB/H/QSNSESS            */
/* 3) Copy QUSRT00L/QATTSYSC/OPSN2API to MYLIB/H/QSNWIN            */
/* 4) Copy QUSRT00L/QATTSYSC/OPSN1API to MYLIB/H/QSNLL            */

#define BOTLINE " Echo lines until:  PF3 - exit"

/* DSM Session Descriptor Structure.                                   */

typedef struct{
    Qsn_Ssn_Desc_T sess_desc;
    char          buffer[300];
}storage_t;

void F3Exit(const Qsn_Ssn_T *Ssn, const Qsn_Inp_Buf_T *Buf, char *action)
{
    *action = '1';
}

int main(void)
{
    int i;
    storage_t    storage;

/* Declarators for declaring windows. Types are from the <qsnsess.h> */
/* header file.                                                        */

    Qsn_Inp_Buf_T    input_buffer = 0;
    Q_Bin4           input_buffer_size = 50;
    char             char_buffer[100];
    Q_Bin4           char_buffer_size;

    Qsn_Ssn_T        session1;
    Qsn_Ssn_Desc_T    *sess_desc = (Qsn_Ssn_Desc_T *) &storage;
    Qsn_Win_Desc_T    win_desc;
    Q_Bin4            win_desc_length = sizeof(Qsn_Win_Desc_T);
    char             *botline = BOTLINE;
    Q_Bin4            botline_len = sizeof(BOTLINE) - 1;
    Q_Bin4            sess_desc_length = sizeof(Qsn_Ssn_Desc_T) +
                                         botline_len;

    Q_Bin4            bytes_read;

/* Initialize Session Descriptor API.                                   */

    QsnInzSsnD( sess_desc, sess_desc_length, NULL);

```

Figure 191. T1520API — ILE C Source to Call an ILE C Procedure (Part 2 of 3)

```

/* Initialize Window Descriptor API.                                     */

QsnInzWinD( &win_desc, win_desc_length, NULL);

sess_desc->cmd_key_desc_line_1_offset = sizeof(Qsn_Ssn_Desc_T);
sess_desc->cmd_key_desc_line_1_len = botline_len;
memcpy( storage.buffer, botline, botline_len );

sess_desc->cmd_key_desc_line_2_offset = sizeof(Qsn_Ssn_Desc_T) +
                                     botline_len;
/* Set up the session type.                                           */

sess_desc->EBCDIC_dsp_cc = '1';
sess_desc->scl_line_dsp = '1';
sess_desc->num_input_line_rows = 1;
sess_desc->wrap = '1';

/* Set up the window size.                                           */

win_desc.top_row      = 3;
win_desc.left_col     = 3;
win_desc.num_rows     = 13;
win_desc.num_cols     = 45;

/* Create a window session.                                           */

sess_desc->cmd_key_action[2] = F3Exit;
session1 = QsnCrtSsn( sess_desc, sess_desc_length,
                     NULL, 0,
                     '1',
                     &win_desc, win_desc_length,
                     NULL, 0,
                     NULL, NULL);
if(input_buffer == 0)
{
    input_buffer = QsnCrtInpBuf(100, 50, 0, NULL, NULL);
}
for (;;)
{

/* Echo lines until F3 is pressed.                                    */

    QsnReadSsnDta(session1, input_buffer, NULL, NULL);
    if (QsnRtvReadAID(input_buffer, NULL, NULL) == QSN_F3)
    {
        break;
    }
}
}

```

Figure 191. T1520API — ILE C Source to Call an ILE C Procedure (Part 3 of 3)

2. To create program T1520API, type:

```
CRTPGM PGM(MYLIB/T1520API) MODULE(MYLIB/T1520API) BNDDIR(QSNAPI)
```

The CRTPGM command creates the program T1520API in library MYLIB. Program T1520API uses DSM Integrated Language Environment bindable API calls to create a window and echo whatever is entered. The prototypes for the

3. To run the program T1520API, type:
CALL PGM(MYLIB/T1520API)

```
AS for iSeries 400 Programming Development Manager (PDM)
.....
: > abc                                     :
: > def                                     :
:                                           :
:                                           :
:                                           :
:                                           :
:                                           :
:                                           :
:                                           :
:                                           :
: ==>                                       :
: Echo lines until: PF3 - exit             :
:                                           :
: .....
Selection or command
==> call pgm(mylib/t1520api)
F3=Exit      F4=Prompt      F9=Retreive      F10=Command entry
F12=Cancel   F18=Change Defaults
```

- The #pragma descriptor for func1() with a #pragma descriptor directive for the function in a header file oper_desc.h.
- An ILE C program that calls func1().
- The ILE C source code of func1() that contains an ILE API that is used to get information from the operational descriptor.

The following shows that the `#pragma` descriptor for `func1` with a `#pragma` descriptor directive for the function in a header file `oper_desc.h`.

```

/* Function prototype in oper_desc.h                                */
int func1( char a[5], char b[5], char *c );
#pragma descriptor(void func1( "", "", "" ))

```

Figure 193. ILE C Source to Generate Operational Descriptors

A function that is named `func1()` is declared. The `#pragma` descriptor for `func1()` specifies that the ILE C compiler must generate string operational descriptors for the three arguments.

The following shows an ILE C program that calls `func1()`. When the function `func1()` is called, the compiler generates operational descriptors for the three arguments that are specified on the call.

```

#include "oper_desc.h"
...
main()
{
    char a[5] = {'s', 't', 'u', 'v', '\0'};
    char *c;
    c = "EFGH";
    ...
    func1(a, "ABCD", c);
}

```

Figure 194. ILE C Source to Call a Function with Operational Descriptors

The following shows the ILE C source code of `func1()` that contains the call to the Integrated Language Environment API. The API is used to determine the string type, length, and maximum length of the string arguments declared in `func1()`. The values for `typeCharZ` and `typeCharV2` are found in the ILE API header file `<leod.h>`.

```

#include <string.h>
#include <stdio.h>
#include <leawi.h>
#include <leod.h>
#include "oper_desc.h"
int func1(char a[5], char b[5], char *c)
{
    int      posn      = 1;
    int      datatype;
    int      currlen;
    int      maxlen;
    _FEEDBACK fc;
    char      *string1;
    /* Call to ILE API CEEGSI to determine string type, length */
    /* and the maximum length.                                */
    CEEGSI(&posn, &datatype, &currlen, &maxlen, &fc);
}

```

Figure 195. ILE C Source to Determine the Strong Arguments in a Function (Part 1 of 2)

```

switch(datatype)
{
    case typeCharZ:
        string1 = a;
        break;
    case typeCharV2:
        string1 = a + 2;
        break;
}
/* Use string1. */
if (!memcmp(string1, "stuv", currlen))
    printf("First 4 characters are the same.\n");
else
    printf("First 4 characters are not the same.\n");
}

```

Figure 195. ILE C Source to Determine the Strong Arguments in a Function (Part 2 of 2)

Calling C++ Programs and Procedures from ILE C

Just as you can call other ILE languages from ILE C, you can call C++ programs and functions as well. You must make sure that you:

1. Declare any C++ functions that you want to call as external.
2. Specify the linkage convention for the call.

Use the `#pragma linkage` and `#pragma argument` directives to specify the linkage convention. See Table 15 on page 326 and “Calling Procedures for ILE C” on page 345 for more information on using these directives.

A C++ program uses the standard OS linkage calling convention. Use `#pragma linkage` to flag the function call as an external program call.

When you call C++ functions, you must make sure that the sender and receiver both agree on the type of parameter being passed, whether it is by pointer or by value, and whether parameters are widened. For example, if the function you are calling was declared as `extern "C nowiden"`, you must use the `#pragma argument(func, nowiden)` directive in the function declaration in ILE C.

You can declare a C++ function as external by either explicitly declaring the function within the C++ code using `extern "C"` or `extern "C nowiden"`. You can add `#ifdef` statements to the function declarations in the header files used by both C and C++ modules, as follows:

These statements are declared with C linkage.

```

#ifdef __cplusplus
extern "C" {
    #endif
    function declarations
#ifdef __cplusplus
}
#endif

```

Calling Procedures for ILE C++

Unlike OPM programs, ILE programs are not limited to using dynamic program calls. ILE programs can also use static procedure calls or procedure pointer calls to call other procedures. Procedure calls are bound calls.

A *static procedure call* is a call to an ILE procedure where the name of the procedure is resolved to an address during binding (static call). Run-time performance of using static procedure calls is faster than run-time performance using dynamic program calls.

```
extern "C" void foo ();
main ()
{
    foo (); // static procedure call
}
```

Note: The term static procedure call does not refer to static storage class but refers to a bound procedure call within a bound module or service program. If the static call is to a procedure written in a language other than C or C++, operational descriptors can be used to resolve the differences in the representation of character strings, if values of this data type are passed as arguments.

Procedure pointer calls provide a way to call a procedure dynamically. You can pass a procedure pointer as a parameter to another procedure which then runs the procedure specified. You can manipulate arrays of procedure names or addresses to dynamically route a procedure call to different procedures. If the called procedure is in the same activation group, the cost of a procedure pointer call is almost identical to the cost of a static procedure call.

Using either type of procedure call, you can call a procedure in a separate module within the same ILE program or service program, or a procedure in a separate ILE service program. Any procedure that can be called using a static procedure call can also be called through a procedure pointer.

Introducing the Call Stack

The *call stack* is a list of call stack entries, in a last-in-first-out (LIFO) order. A *call stack entry* is a call to a program or procedure. There is one call stack per job.

When an ILE program is called, the program entry procedure is first added to the call stack. After the program entry procedure is called, control is given to the main entry point in the program (`main()` for C or C++) which is pushed onto the stack.

Figure 196 on page 353 shows a call stack for an program consisting of an OPM program which calls an ILE program consisting of two modules: a C++ module containing the program entry procedure and the associated user entry procedure, and a C module containing a regular procedure. The most recent entry is at the bottom of the stack.

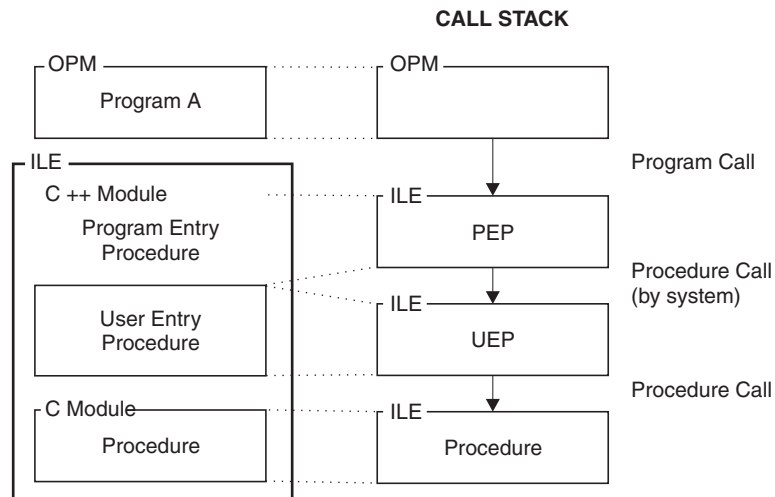


Figure 196. Program and Procedure Calls on the Call Stack

Note: In a dynamic program call, the calls to the program entry procedure and the user entry procedure (UEP) occur together, since the call to the UEP is automatic. In later diagrams involving the call stack, the two steps of a dynamic program call are combined.

An ILE C++ procedure which is on the call stack can be called before it returns to its caller. This is a *recursive* procedure call. This is different from an ILE COBOL procedure which when on the call stack cannot be called until it returns to its caller. This is a *non-recursive* procedure call. Therefore, be careful not to call from ILE C++ procedures another ILE COBOL procedure which might call an already active ILE COBOL procedure.

Assume that procedure A is an ILE C++ procedure, procedure B and C are ILE COBOL procedures, and that these procedures are in the same program. If procedure A calls procedure B, then procedure B can call neither procedure A nor B. If procedure B returns and if procedure A then calls procedure C, procedure C can call procedure B but not procedure A or C. See Figure 197.

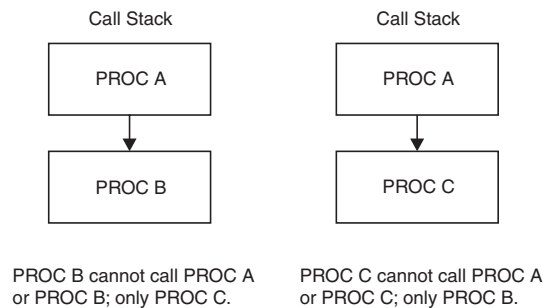


Figure 197. ILE C++ Procedures Cannot Call Other Active ILE COBOL Procedures

OPM COBOL programs already on the call stack cannot be called.

Calling a Program Using a Linkage Specification

You can call OPM, ILE, or EPM programs from a C++ program. OPM, ILE or EPM programs can also call a C++ program.

C++ provides a linkage specification to enable dynamic program calls and sharing of data between them. The syntax is:

```
►►extern—"string-literal"—{ declaration-list }►►  
  
►►extern—"string-literal"—declaration►►
```

The *"string-literal"* is used to specify the linkage associated with a particular function. The string literals used in linkage specifications are case-insensitive. The valid string literals for the linkage specification to call programs are:

"OS" OS linkage call

"OS nowiden"

OS linkage call without widened parameters. See "OS Linkage" on page 358 for details.

If you want a C++ program to call an ILE, OPM, or EPM program (*PGM), use the extern "OS" linkage specification in your C++ source to tell the compiler that the called program is an external program, not a bound ILE procedure. For example, if you want a C++ program to call an OPM COBOL program (*PGM) this extern "OS" linkage specification in your C++ source tells the compiler that COBOL_PGM is an external program, not a bound ILE procedure.

```
extern "OS" void COBOL_PGM(void);
```

If you want an ILE, OPM or EPM program to call a C++ program, use the ILE, OPM, or EPM language-specific call statement.

Calling an ILE Procedure Using a Linkage Specification

Unlike OPM programs, ILE programs are not limited to using dynamic program calls. ILE programs can also use static procedure calls or procedure pointer calls.

Notes:

1. ILE procedures within an *activated* ILE program can be accessed using static procedure calls or procedure pointer calls. New ILE programs that have not been activated yet must be called by a dynamic program call. There are other situations in which dynamic calls occur.
2. EPM C and Pascal procedures or functions cannot call C++ procedures.
3. The term procedure in ILE is similar to the term function in C++. A C++ function is an ILE procedure.

C++ provides a linkage specification to enable procedure calls and the sharing of data between the C++ caller and the called procedure. See "Calling a Program Using a Linkage Specification" on page 353 for the syntax.

The valid string literals for the linkage specification to call ILE procedures are:

Linkage Specification	Type of Procedure Called
"C++"	ILE C++ procedure (default)
"C"	ILE C procedure
"C nowiden"	ILE C procedure without widened parameters
"RPG"	ILE RPG procedure

"COBOL"	ILE COBOL procedure
"CL"	ILE CL procedure
"ILE"	General ILE function call
"ILE nowiden"	ILE function call without widened parameters
"VREF"	ILE function call with pointers in temporary storage. (Behaves the same as a regular call although parameters are passed to the function as if they were by reference.)
"VREF nowiden"	Same as "VREF" without widened parameters

Passing Parameters

To share data between programs or between procedures, you need to pass the called program or procedure parameters which both programs can use. In C++ you use the linkage specification to tell the compiler which parameter passing convention to use on the external call.

When passing parameters from C++ to a different high-level language (HLL) consider:

- Parameter passing style of the HLLs

Each HLL has its own way of passing parameters. Parameters can be passed as a pointer to the parameter value, to a copy of the value, or to the value itself. C++ passes parameters in all three ways. See "Using Default Parameter Passing Styles" on page 359 for information on these styles.

- Interlanguage data compatibility

Different HLLs support different ways of representing data. Pass only parameters which have a data type common to the calling and called program or procedure. If you are not sure of the exact format of the data that is passed to your program, you may specify to the users of your procedure that an operational descriptor can be passed to provide additional information regarding the format of the passed parameters. See "Using Operational Descriptors" on page 360.

Passing Parameters in C++

Table 17 shows the effect of using the using different linkage types on passing parameters:

Table 17. Effects of Various Linkage Specifications

Linkage	Name Mangled	Parameter Passing	Parameter Widening	Comments
"C++"	Yes	C++	No	This is the default.
"C"	No	C++	Yes	Used to call a function (procedure) written in ILE C.
"C nowiden"	No	C++	No	Used to call a function (procedure) written in ILE C.
"OS"	No	OS	Yes	Used to call an external program written in any OPM/EPM/ILE language.

Table 17. Effects of Various Linkage Specifications (continued)

Linkage	Name Mangled	Parameter Passing	Parameter Widening	Comments
"OS nowiden"	No	OS	No	Used to call an external program written in any OPM/EPM/ILE language.
"RPG"	No	OS	No	Used to call a procedure written in ILE RPG. Parameters with address types (pointer or reference) are passed by value directly. All other parameters are passed by value indirectly.
"COBOL"	No	OS	No	Used to call a procedure written in ILE COBOL. Parameters with address types (pointer or reference) are passed by value directly. All other parameters are passed by value indirectly.
"CL"	No	OS	No	Used to call a procedure written in ILE CL. Parameters with address types (pointer or reference) are passed by value directly. All other parameters are passed by value indirectly.
"ILE"	No	OS	Yes	Used to call a procedure written in an ILE language. Identical to RPG, COBOL, and CL specifications. If the particular language in which the function was written is unknown to the programmer, use this linkage. If you have C code that uses the #pragma argument directive and you plan to port this code to C++ then use the extern "ILE" linkage specification.
"ILE nowiden"	No	OS	No	Used to call a procedure written in an ILE language.
"VREF"	No	OS	Yes	Used to call a procedure written in an ILE language. Parameters are passed by value indirectly.
"VREF nowiden"	No	OS	No	Used to call a procedure written in an ILE language. Parameters are passed by value indirectly.

The extern keyword followed by the string literals "RPG", "COBOL", or "CL" are used to specify that the function has "ILE" linkage. These string literals perform the same function as the **#pragma argument** directive in ILE C. The "VREF" linkage also performs the same as the *VREF* parameter on the **#pragma argument** directive.

The *string-literal* is case-insensitive: extern "OS NOWIDEN", extern "OS nowiden", and extern "os NoWiden", although different in case are all handled in the same way.

The name of the function must follow the naming conventions of the other language. For OS linkage specifications, the program name and all iSeries objects must be uppercase.

A typedef of a function can be declared to have linkage information. After the typedef is declared, it can be used to declare functions of a particular linkage. The typedef declaration must be enclosed by braces {}.

A function cannot be assigned directly to a pointer to a function with a different linkage. A type cast may be used to make this assignment possible. Type casting helps you eliminate parameter mismatching problems without excess constraint.

To override a function but not to override extern "OS" use a type cast:

```
extern "ILE"
{
    typedef void (*ILE) ();
}
extern "C++"
{
    typedef void (*CPP) ();
}
ILE pILE;
CPP pCPP = (CPP) pILE;
```

Functions that take function pointers as parameters may not be overloaded based on the linkage of the function pointers:

```
// Using the typedef declarations above
void foo (OS);
void foo (CPP); // undefined behavior, foo already declared
```

Functions that are defined with non-C++ linkage specifications accept parameters using the appropriate convention for that linkage. You do not need to widen parameters:

```
extern "C" void foo (char); // chars are widened in C
// In another compilation unit we then have
extern "C" void foo (char c) // this parameter is correctly widened
{
    // implementation of foo (char);
}
```

Attempting to define a function with either extern "OS", extern "OS nowiden", or extern "builtin" linkage results in undefined behavior:

```
extern "OS" void FOOPGM (char); // declaration: OK
extern "OS" void FOOPGM (char c) // definition: undefined behavior
{
    // implementation of FOOPGM
}
```

Functions F001(), F002(), and F003() are all declared as OS linkage functions. Functions F001() and F002() are declared by using the declaration-list syntax. F003() is declared by using the simple declaration.

```
extern "OS" {
    void F001 (char, char *);
    void F002 (int, int *);
}
extern "OS" double F003 (double, double *);
```

The declaration of a function pointer is:

```
extern "OS" {
    typedef void (*fp) (char);
}
fp F00;
```

Function F00() is declared to be a function pointer of type fp.

Data objects can be declared inside the extern linkage declaration:

```
extern "OS" {
    int a1;
}
extern "OS" int a2;
```

Variable a1 is defined while variable a2 is only declared.

The widening rules for all the linkage specifications shown in Table 17 on page 355 are:

- Any data type that is smaller than int is widened to int
- float is widened to double
- Address and Data pointers are not widened
- struct has the same structure and information as the struct parameter passed

In C++ linkage specifications, function identifiers are mangled. In all other linkage specifications, all function identifiers are identical to the exported names unless changed by the **#pragma map** directive.

OS Linkage

The extern specifier followed by the *string-literal* "OS" or the *string-literal* "OS nowiden" is used to declare external programs. These programs may then be called in the same way as a regular function.

When an OS linkage function is called from a C++ program, the compiler generates code and performs the following tasks in sequence:

1. If extern "OS" is used, then the parameters and return value are widened.
If extern "OS nowiden" is used, then the parameters and return value passed between programs are not widened.
2. Parameters that are passed by value are copied to temporary variables and the addresses of the temporary variables are passed to the called program.
If a temporary variable is created for a structure, the temporary variable has the same structure and information as the struct parameter passed.
3. Parameters that were passed by reference are still passed by reference.
4. Arrays and pointers are passed by reference.
5. If the argument you are passing is an array name or a pointer, then the argument is passed directly, and a temporary variable is not created. The data referenced by the array or pointer can be changed by the called program.
6. The function name is not mangled.

The program name that the C++ dynamic program calls must be in uppercase. You can use the **#pragma map** directive to map an internal identifier longer than 10 characters to an OS/400-compliant object name (10 characters or less) in your program. See "Changing the Names of Programs and Procedures" on page 371.

The return code for the dynamic program call can be retrieved by declaring the program to return an integer:

```
extern "OS" int PGMNAME(void);
```

The value returned on the call is the return code for the dynamic program call. If the program being called is a C++ program, this return code can be accessed using the `_LANGUAGE_RETURN_CODE` macro defined in the header file `<mlib.h>`. A C++ program returns four bytes in the `_LANGUAGE_RETURN_CODE`. If the program being called is an EPM or OPM program, this return code can be accessed using the `iSeries Retrieve Job Attributes (RTVJOBA)` command.

When a function is called from an OS linkage function pointer, the compiler generates the same code sequence it does when calling an OS linkage function.

Nonpointer arguments are passed by value reference, and changes made to the variables in the called program are not reflected in the calling C++ program.

C Linkage

Specifying C linkage for a function tells the compiler:

- Parameters are passed using C++ conventions
- Parameters for functions declared with `extern "C"` are widened
- The function name is not mangled

The `extern` keyword followed by the *string-literal* `"C"` or the *string-literal* `"C nowiden"` is used to specify that the function is declared to have "C" linkage instead of "C++" linkage.

ILE, CL, COBOL, and RPG Linkage

Specifying an ILE, CL, COBOL, or RPG linkage for a function tells the compiler:

- Arguments passed as values or nonpointer arguments are copied to temporary variables and the addresses of the temporary variables are passed to the called program
- Pointer arguments are passed directly to the called program
- If `extern "ILE nowiden"` is used, then the parameters and return value passed between programs are not widened; specifying any other ILE linkage widens the parameters
- Function names are not mangled

VREF Linkage

Specifying a VREF linkage is identical to specifying an ILE linkage except that pointer parameters are stored in a temporary variable and the address of the temporary variable is passed as the actual argument.

Using Default Parameter Passing Styles

ILE C++ uses the same calling mechanisms for calling any ILE HLL program or procedure: `extern` linkage specification. ILE C++ passes and receives parameters using three passing methods: by value directly, by value indirectly, and by reference. Other ILE languages may have different methods of passing data. See Table 18 on page 360.

by value, directly

The value of the data object is placed directly into the argument list.

by value, indirectly

The value of the data object is copied to a temporary location. The address of the copy, a pointer, is placed into the argument list.

by reference

A pointer to the data object is placed into the argument list. Changes made by the called procedure to the argument are reflected in the calling procedure.

Table 18 shows the common parameter-passing methods for the ILE programs.

Table 18. Default Argument Passing Style for ILE Programs

ILE HLL	Pass Argument	Receive Argument
ILE C	by value, directly or by reference	by value, directly or by reference
ILE C++	by value, directly or by value, indirectly or by reference	by value, directly, or by value, indirectly or by reference
ILE COBOL	by reference or by value, indirectly	by reference or by value, indirectly
ILE RPG	by reference	by reference
ILE CL	by reference	by reference

Table 19 shows the common parameter passing methods for the ILE procedures.

Table 19. Default Argument Passing Style for ILE Procedures

ILE HLL	Pass Argument	Receive Argument
ILE C	by value, directly	by value, directly
ILE C++	by value, directly or by value, indirectly or by reference	by value, directly, or by value, indirectly or by reference
ILE COBOL	by reference or by value, indirectly	by reference
ILE RPG	by reference	by reference
ILE CL	by reference	by reference

To pass or receive parameters to or from procedure calls involving other ILE languages, especially ILE C, ILE RPG, or ILE COBOL, you must ensure that the other procedure is set up to accept data by reference.

Using Operational Descriptors

To pass a parameter to a procedure even though the data type is not precisely known to the called procedure you can use operational descriptors. *Operational descriptors* provide descriptive information to the called procedure regarding the form of the argument. This information allows the procedure to properly interpret the passed parameter. Only use operational descriptors when they are expected by the called procedure.

Note: The C++ compiler supports operational descriptors for describing null-terminated strings. A character string in C++ is defined by: `char string_name[n]`, `char * string_name`, or *string-literal*.

C++ defines a string as a contiguous sequence of characters terminated by and including the first null character. In another language, a string may be defined as consisting of a length specifier and a character sequence. When passing a string from a C++ function to a function written in another language, an operational descriptor can be provided with the argument to allow the called function to determine the length and type of the string being passed.

To use operational descriptors, you specify a **#pragma descriptor** directive in your source to identify functions whose arguments have operational descriptors. Operational descriptors are then built by the calling procedure and passed as hidden arguments to the called procedure.

The syntax is:

►—#—pragma—descriptor—(—void function_name(—od_specifiers)—)————►

See the *VisualAge for C++ for AS/400 C++ Language Reference* for information on operational descriptors.

Understanding Data-Type Compatibility

Each high-level language has different data types. When you want to pass data between programs written in different languages, you must be aware of these differences.

Some data types in the ILE C++ programming language have no direct equivalent in other languages. You can simulate data types in other languages using ILE C++ data types.

Note: No data-type compatibility tables are shown for C. You can use the C++ tables since C and C++ data types are the same except for the packed decimal data type. In C++ the packed decimal data type is implemented as a class. The packed decimal data type in ILE C and the binary coded decimal class in C++ are binary compatible.

Table 20 shows the ILE C++ data type compatibility with ILE RPG.

Table 20. ILE C++ Data-Type Compatibility with ILE RPG

ILE C++ declaration in prototype	ILE RPG D spec, columns 33 to 39	Length	Comments
char[n]	nA	n	An array of characters where n=1 to 32766
char *	*	16	A pointer
char	1A	1	An indicator which is a variable starting with *IN
char[n]	nS 0	n	A zoned decimal
char[2n]	nG	2n	A graphic added
char[2n+2]	Not supported	2n+2	A graphic data type
_Packed struct {short i; char[n]}	data structure	n+2	A variable length field where i is the intended length and n is the maximum length
char[n]	D	8, 10	A date field
char[n]	T	8	A time field
char[n]	Z	26	A timestamp field
short int	5I 0	2	An integer field
short unsigned int	5U 0	2	An unsigned integer field
int	10I 0	4	An integer field

Table 20. ILE C++ Data-Type Compatibility with ILE RPG (continued)

ILE C++ declaration in prototype	ILE RPG D spec, columns 33 to 39	Length	Comments
unsigned int	10U 0	4	An unsigned integer field
long int	10I 0	4	An integer field
long unsigned int	10I 0	4	An unsigned integer field
struct {unsigned int : n}x;	Not supported	1, 2, 4	A 4-byte unsigned integer, a bitfield
float	Not supported	4	A 4-byte floating point
double	Not supported	8	An 8-byte floating point
long double	Not supported	8	An 8-byte floating point
enum	Not supported	1, 2, 4	Enumeration
_DecimalT<n,p>	nP p	n/2+1	A packed decimal. n must be less than or equal to 30. In C++, this is a binary coded decimal class and not a data type.
union.element	<type> with keyword OVERLAY(longest field)	length of longest union member	An element of a union
data_type[n]	<type> with keyword DIM(n)	16	An array to which C++ passes a pointer
struct or class	data structure	n	A structure. The structure must be packed. ¹ Structures passed should be passed as a pointer to the structure if you want to change the contents of the structure.
pointer to function	* with keyword PROCPTR	16	A 16-byte pointer
Note: ¹ All structures must be packed. Classes with virtual functions cannot be directly represented in any other language. Nested structures must be explicitly packed. In C, all nested structures are packed.			

Table 21 shows the ILE C++ data-type compatibility with ILE COBOL.

Table 21. ILE C++ Data-Type Compatibility with ILE COBOL

ILE C++ declaration in prototype	ILE COBOL LINKAGE SECTION	Length	Comments
char[n] char *	PIC X(n).	n	An array of characters where n=1 to 3,000,000
char	PIC 1 INDIC ..	1	An indicator
char[n]	PIC S9(n) DISPLAY.	n	A zoned decimal
wchar_t[n]	PIC G(n) DISPLAY.	2n	A graphic data type
_Packed struct {short i; char[n]}	05 VL-FIELD. 10 i PIC S9(4) COMP-4. 10 data PIC X(n).	n+2	A variable length field where i is the intended length and n is the maximum length.

Table 21. ILE C++ Data-Type Compatibility with ILE COBOL (continued)

ILE C++ declaration in prototype	ILE COBOL LINKAGE SECTION	Length	Comments
char[n]	PIC X(n) .	6	A date field
char[n]	PIC X(n) .	5	A day field
char	PIC X .	1	A day-of-week-field
char[n]	PIC X(n) .	8	A time field
char[n]	PIC X(n) .	26	A time stamp field
short int	PIC S9(4) COMP-4 .	2	A 2-byte signed integer with a range of -9999 to +9999
short int	PIC S9(4) BINARY .	2	A 2-byte signed integer with a range of -9999 to +9999
int	PIC S9(9) COMP-4 .	4	A 4-byte signed integer with a range of -999999999 to +999999999
int	PIC S9(9) BINARY .	4	A 4-byte signed integer with a range of -999999999 to +999999999
int	USAGE IS INDEX	4	A 4-byte integer
long int	PIC S9(9) COMP-4 .	4	A 4-byte signed integer with a range of -999999999 to +999999999
long int	PIC S9(9) BINARY .	4	A 4-byte signed integer with a range of -999999999 to +999999999
struct {unsigned int : n}x;	PIC 9(9) COMP-4 . PIC X(4) .	1, 2, 4	Bitfields can be manipulated using hex literals
float	Not supported	4	A 4-byte floating point
double	Not supported	8	An 8-byte floating point
long double	Not supported	8	An 8-byte floating point
enum	Not supported	1, 2, 4	Enumeration
*	USAGE IS POINTER	16	A pointer
_DecimalT<n,p>	PIC S9(n-p)V9(p) COMP-3 .	n/2+1	A packed decimal. In C++, this is a binary coded decimal class and not a data type.
_DecimalT<n,p>	PIC S9(n-p) 9(p) PACKED-DECIMAL .	n/2+1	A packed decimal. In C++ this is a binary coded decimal class not a data type.
union.element	REDEFINES	length of longest union member	An element of a union
data_type[n]	OCCURS	n times the length of the data type	An array to which C++ passes a pointer
struct or class	OCCURS ...DEPENDING ON	variable	A structure. The structure must be packed. ¹ Structures passed should be passed as a pointer to the structure if you want to change the contents of the structure.

Table 21. ILE C++ Data-Type Compatibility with ILE COBOL (continued)

ILE C++ declaration in prototype	ILE COBOL LINKAGE SECTION	Length	Comments
struct or class	01 record 05 field1 05 field2	n	A structure. The structure must be packed. ¹ Structures passed should be passed as a pointer to the structure if you want to change the contents of the structure.
pointer to function	PROCEDURE-POINTER	16	A 16-byte pointer to a procedure
Not supported.	PIC S9(18) COMP-4.	8	An 8-byte integer
Not supported.	PIC S9(18) BINARY.	8	An 8-byte integer
Note: ¹ All structures must be packed. Classes with virtual functions cannot be directly represented in any other language. Nested structures must be explicitly packed. In C, all nested structures are packed.			

Table 22 shows the ILE C++ data-type compatibility with ILE CL.

Table 22. ILE C++ Data-Type Compatibility with ILE CL

ILE C++ declaration in prototype	CL	Length	Comments
char[n] char *	*CHAR LEN(&N)	n	An array of characters where n=1 to 32766. A null-terminated string. CHGVAR &V1 VALUE(&V *TCAT X'00')
char	*LGL	1	Holds '1' or '0'
_Packed struct {short i; char[n]}	Not supported	n+2	A variable length field where i is the intended length and n is the maximum length.
integer types	Not supported	1, 2, 4	A 1, 2 or 4 byte signed or unsigned integer
float constants	CL constants only	4	A 4 or 8 byte floating point
_DecimalT<n,p>	*DEC	n/2+1	A packed decimal. The limit of n is 15 and p is 9. In C++, this is a binary coded decimal class and not a data type.
union.element	Not supported	length of longest union member	An element of a union
struct or class	Not supported	n	A structure. The structure must be packed. ¹ Structures passed should be passed as a pointer to the structure if you want to change the contents of the structure.
pointer to function	Not supported	16	A 16-byte pointer

Table 23 shows the ILE C++ data-type compatibility with OPM RPG.

Table 23. ILE C++ Data-Type Compatibility with OPM RPG

ILE C++ declaration in prototype	OPM RPG I spec, DS subfield columns spec	Length	Comments
char[n]	1 10	n	An array of characters where n=1 to 32766
char	*INxxxx	1	An indicator which is a variable starting with *IN
char[n]	1 nd (d>=0)	n	A zoned decimal The limit of n is 30
char[2n+2]	Not supported	2n+2	A graphic data type
_Packed struct {short i; char[n]}	Data structure	n+2	A variable length field where i is the intended length and n is the maximum length.
char[n]	char	6, 8, 10	A date field
char[n]	char	8	A time field
char[n]	char	26	A time stamp field.
short int	B 1 20	2	A 2-byte signed integer with a range of -9999 to +9999
int	B 1 40	4	A 4-byte signed integer with a range of -999999999 to +999999999
long int	B 1 40	4	A 4-byte signed integer with a range of -999999999 to +999999999
struct {unsigned int : n}x;	Not supported	1, 2, 4	A 4-byte unsigned integer, a bitfield
float	Not supported	4	A 4-byte floating point
double	Not supported	8	An 8-byte floating point
long double	Not supported	8	An 8-byte floating point
enum	Not supported	1, 2, 4	Enumeration
*	Not supported	16	A pointer
_DecimalT<n,p>	P 1 n/2+1d	n/2+1	A packed decimal. n must be less than or equal to 30. In C++, this is a binary coded decimal class and not a data type.
union.element	data structure subfield	length of longest union member	An element of a union
data_type[n]	E-SPEC array	16	An array to which C++ passes a pointer
struct or class	data structure	n	A structure. The structure must be packed. ¹ Structures passed should be passed as a pointer to the structure if you want to change the contents of the structure.
pointer to function	Not supported	16	A 16-byte pointer

Table 23. ILE C++ Data-Type Compatibility with OPM RPG (continued)

ILE C++ declaration in prototype	OPM RPG I spec, DS subfield columns spec	Length	Comments
Note: ¹ All structures must be packed. Classes with virtual functions cannot be directly represented in any other language. Nested structures must be explicitly packed. In C, all nested structures are packed.			

Table 24 shows the ILE C++ data-type compatibility with OPM COBOL.

Table 24. ILE C++ Data-Type Compatibility with OPM COBOL

ILE C++ declaration in prototype	OPM COBOL LINKAGE SECTION	Length	Comments
char[n] char *	PIC X(n).	n	An array of characters where n=1 to 3,000,000
char	PIC 1 INDIC ..	1	An indicator
char[n]	PIC S9(n) USAGE IS DISPLAY	n	A zoned decimal The limit of n is 18.
_Packed struct {short i; char[n]}	05 VL-FIELD. 10 i PIC S9(4) COMP-4. 10 data PIC X(n).	n+2	A variable length field where i is the intended length and n is the maximum length
char[n]	PIC X(n).	6, 8, 10	A date field
char[n]	PIC X(n).	8	A time field
char[n]	PIC X(n).	26	A time stamp field
short int	PIC S9(4) COMP-4.	2	A 2-byte signed integer with a range of -9999 to +9999.
int	PIC S9(9) COMP-4.	4	A 4-byte signed integer with a range of -999999999 to +999999999
long int	PIC S9(9) COMP-4.	4	A 4-byte signed integer with a range of -999999999 to +999999999
struct {unsigned int : n}x;	PIC 9(9) COMP-4. PIC X(4).	1, 2, 4	Bitfields can be manipulated using hex literals.
float	Not supported	4	A 4-byte floating point
double	Not supported	8	An 8-byte floating point
long double	Not supported	8	An 8-byte floating point
enum	Not supported	1, 2, 4	Enumeration
*	USAGE IS POINTER	16	A pointer
_DecimalT<n,p>	PIC S9(n-p)V9(p) COMP-3.	n/2+1	A packed decimal The limits of n and p are 18. In C++, this is a binary coded decimal class and not a data type.
union.element	REDEFINES	length of longest union member	An element of a union

Table 24. ILE C++ Data-Type Compatibility with OPM COBOL (continued)

ILE C++ declaration in prototype	OPM COBOL LINKAGE SECTION	Length	Comments
data_type[n]	OCCURS	n times the length of the data type	An array to which C++ passes a pointer
struct or class	OCCURS ...DEPENDING ON	variable	A structure. The structure must be packed. ¹ Structures passed should be passed as a pointer to the structure if you want to change the contents of the structure.
struct or class	01 record	n	A structure. The structure must be packed. ¹ Structures passed should be passed as a pointer to the structure if you want to change the contents of the structure.
pointer to function	Not supported	16	A 16-byte pointer
Not supported.	PIC S9(18) COMP-4.	8	An 8-byte integer
Note: ¹ All structures must be packed. Classes with virtual functions cannot be directly represented in any other language. Nested structures must be explicitly packed. In C all nested structures are packed.			

Table 25 shows the ILE C++ data-type compatibility with CL.

Table 25. ILE C++ Data-Type Compatibility with CL

ILE C++ declaration in prototype	CL	Length	Comments
char[n] char *	*CHAR LEN(&N)	n	An array of characters where n=1 to 32766. A null-terminated string. CHGVAR &V1 VALUE(&V *TCAT X'00') where &V1 is one byte bigger than &V. The limit of n is 9999.
char	*LGL	1	Holds '1' or '0'
_Packed struct {short i; char[n]}	Not supported	n+2	A variable length field where i is the intended length and n is the maximum length.
integer types	Not supported	1, 2, 4	A 1, 2 or 4 byte signed or unsigned integer.
float constants	CL constants only	4	A 4 or 8 byte floating point
_DecimalT<n,p>	*DEC	n/2+1	A packed decimal. The limit of n is 15 and p is 9. In C++, this is a binary coded decimal class and not a data type.

Table 25. ILE C++ Data-Type Compatibility with CL (continued)

ILE C++ declaration in prototype	CL	Length	Comments
union.element	Not supported	length of longest union member	An element of a union
struct or class	Not supported	n	A structure. The structure must be packed. ¹ Structures passed should be passed as a pointer to the structure if you want to change the contents of the structure.
pointer to function	Not supported	16	A 16-byte pointer

Passing Arguments from a CL Program to an ILE C++ Program

Table 26 shows how arguments are passed from a command line CL call to an ILE C++ program.

Table 26. Arguments Passed From a Command Line CL Call to an ILE C++ Program

Command Line Argument	Argv Array	ILE C++ Arguments
	argv[0]	"LIB/PGMNAME"
	argv[1..255]	normal parameters
'123.4'	argv[1]	"123.4"
123.4	argv[2]	__D("0000000123.40000")
'Hi'	argv[3]	"Hi "
Lo	argv[4]	"L0"
'1'	argv[5]	"1"

A CL character array is not null-terminated when it is passed to another program. A C++ program that receives such an argument from a CL program should not expect the strings to be null-terminated. You can use the QCAPEXC to ensure that all the arguments are null-terminated.

Table 27 shows how CL constants are passed from a compiled CL program to an ILE C++ program.

Table 27. CL Constants Passed from a Compiled CL Program to an ILE C++ Program

Compile CL Program Argument	Argv Array	ILE C++ Arguments
	argv[0]	"LIB/PGMNAME"
	argv[1..255]	normal parameters
'123.4'	argv[1]	"123.4"
123.4	argv[2]	__D("0000000123.40000")
'Hi'	argv[3]	"Hi "
Lo	argv[4]	"L0"

Table 27. CL Constants Passed from a Compiled CL Program to an ILE C++ Program (continued)

Compile CL Program Argument	Argv Array	ILE C++ Arguments
'1'	argv[5]	"1"

A command processing program (CPP) passes CL constants as defined in Table 27 on page 368. You can create your own CL command with the Create Command (CRTCMD) command and define an ILE C++ program as the command processing program.

Table 28 shows how CL variables are passed from a compiled CL program to an ILE C++ program. All arguments are passed by reference from CL to C++.

Table 28. CL Variables Passed from a Compiled CL Program to an ILE C++ Program

CL Variables	C++ Arguments
DCL VAR(&v) TYPE(*CHAR) LEN(10) VALUE('123.4')	"123.4"
DCL VAR(&d) TYPE(*DEC) LEN(10 1) VALUE(123.4)	__D("0000000123.40000")
DCL VAR(&h) TYPE(*CHAR) LEN(10) VALUE('Hi')	"Hi "
DCL VAR(&i) TYPE(*CHAR) LEN(10) VALUE(Lo)	"L0"
DCL VAR(&j) TYPE(*LGL) LEN(1) VALUE('1')	"1"

CL variables and numeric literals are not passed to an ILE C++ program with null-terminated strings. Character literals and logical literals are passed as null-terminated strings but are not padded with blanks. Numeric literals such as packed decimals are passed as 15,5 (8 bytes).

The CL program CLPROG1 passes the parameters v, d, h, i, j to an ILE C++ program MYPROG1.

The parameters are null-terminated within the the CL program CLPROG1. They are passed by reference. All incoming arguments to MYPROG1 are pointers.

```

/* CLPROG1
PGM          PARM(&V &D &H &I &J)
  DCL          VAR(&V) TYPE(*CHAR) LEN(10)
  DCL          VAR(&VOUT) TYPE(*CHAR) LEN(11)
  DCL          VAR(&D) TYPE(*DEC) LEN(10 1)
  DCL          VAR(&H) TYPE(*CHAR) LEN(10)
  DCL          VAR(&HOUT) TYPE(*CHAR) LEN(11)
  DCL          VAR(&I) TYPE(*CHAR) LEN(10)
  DCL          VAR(&IOUT) TYPE(*CHAR) LEN(11)
  DCL          VAR(&J) TYPE(*LGL) LEN(1)
  DCL          VAR(&JOUT) TYPE(*LGL) LEN(2)
  DCL          VAR(&NULL) TYPE(*CHAR) LEN(1) VALUE(X'00')
/* ADD NULL TERMINATOR FOR THE C++ PROGRAM */
CHGVAR        VAR(&VOUT) VALUE(&V *TCAT &NULL)
CHGVAR        VAR(&HOUT) VALUE(&V *TCAT &NULL)
CHGVAR        VAR(&IOUT) VALUE(&V *TCAT &NULL)
CHGVAR        VAR(&JOUT) VALUE(&V *TCAT &NULL)
CALL          PGM(MYPROG1) PARM(&VOUT &D &HOUT &IOUT &JOUT)
ENDPGM

```

The CL program CLPROG1 receives its input values from a CL Command Prompt MYCMD1 which prompts the user to input the desired values. The source code for MYCMD1 is:

```

CMD      PROMPT('ENTER VALUES')
PARM     KWD(V) TYPE(*CHAR) LEN(10) +
          PROMPT('1ST VALUE')
PARM     KWD(D) TYPE(*DEC) LEN(10 2) +
          PROMPT('2ND VALUE')
PARM     KWD(H) TYPE(*CHAR) LEN(10) +
          PROMPT('3RD VALUE')
PARM     KWD(I) TYPE(*CHAR) LEN(1) +
          PROMPT('4TH VALUE')
PARM     KWD(J) TYPE(*LGL) LEN(10 2) +
          PROMPT('5TH VALUE')

```

After the CL program CLPROG1 has received the user input from the command prompt MYCMD1, it passes the input values on to a C++ program MYPROG1. The source code for this program is contained in myprog1.cpp:

```

// myprog1.cpp

#include <iostream.h>
#include <stdlib.h>
#include <string.h>
#include <bcd.h>

// Arguments are received by reference from CL program CLPROG1
// Incoming arguments are all pointers

int main(int argc, char *argv[])
{
    char          *v;
    char          *h;
    char          *i;
    char          *j;
    _DecimalT <10, 1> d;

    v = argv[1];
    d = *((_DecimalT <10,1> *) argv[2]);
    h = argv[3];
    i = argv[4];
    j = argv[5];
    cout << " v= " << v
         << " d= " << d
         << " h= " << h
         << " i= " << i
         << " j= " << j
         << endl;
}

```

If the CL program CLPROG1 passes the following parameters to the C++ program MYPROG1:

'123.4', 123.4, 'Hi', L0, and '1'

the output from program MYPROG1 is:

```

v= 123.4    HI      L0      1 d= 123.4 h= HI      L0      1
i= L0      1 j= 1
Press ENTER to end terminal session.

```

Changing the Names of Programs and Procedures

You may want to change the name of an ILE procedure to make it more descriptive, and to make the ILE procedure easy to identify for maintenance purposes. An ILE procedure name QRZ1233 could be renamed to checkmod. You may want to change the name of a program that contains an illegal character; A-B is not a valid name in a C++ program.

You can use the **#pragma map** directive to map an internal identifier to an OS/400-compliant name (10 characters or less for program names and 1 or more characters for ILE procedure names) in your program.

The syntax is:

```
▶▶ #pragma map (—identifier , —"name" —) ▶▶
▶▶ #pragma map ▶▶
▶▶ (—function-or-operator-identifier (—argument-list —) , —"name" —) ▶▶
```

This pragma tells the compiler that all references to identifier are to be converted to "name". The pragma can appear anywhere in the source file within a single compilation unit. It can appear before any declaration or definition of the named object, function or operator. The identifiers appearing in the directive, including any type names used in the prototype argument list, are resolved as though the directive had appeared at file scope, independent of its actual point of occurrence:

```
int func(int);

class X
{
public:
    void func(void);
    #pragma map(func, "funcname1") //maps ::func
    #pragma map(X::func, "funcname2") //maps X::func
};
```

There are two functions named func() in this code. One is a regular function, prototyped int func(int) and the other is a class member function void func(void). To avoid confusion, they are renamed funcname1, and funcname2 using the **#pragma map** directive.

Mapping can be based on parameter type as well as scope. For example, void func(int); void func(char); #pragma map(func(int), "intFunc") #pragma map(func(char), "charFunc"). This does not work with *PRV option.

Creating C++ Classes for Use in ILE

You can access existing C++ classes from other languages such as ILE C, but you need to write your own functions to display and manipulate the data members of these classes.

A shared C/C++ header for class MyClass might look like the following:

```
/* myclass.h */
#ifdef __cplusplus
class MyClass {
```

```

public:
MyClass()
{
n = new int[100];
}
~MyClass()
{
delete [] n;
}
int &operator[] (int i)
{
return n[i];
}
private:
int *n;
};
#else
struct MyClass;
MyClass *createMyClass();
void destroyMyClass(MyClass*);
int *MyClassIndex(int);
#endif

```

Mapping a C++ Class to a C Structure

A C++ class without virtual functions can be mapped to a corresponding C structure, but there are fundamental differences between both data types. The C++ class contains data members and member functions to access and manipulate these data members. The corresponding C structure contains only the data members, but not the member functions contained in the C++ class.

The class Class1 in C++ is:

```

class Class1
{
public:
    int m1;
    int m2;
    int m3;
    f1();
    f2();
    f3();
};

```

The corresponding C structure is:

```

struct Class1
{
    int m1;
    int m2;
    int m3;
};

```

To access a C++ class from a C program you need to write your own functions to inspect and manipulate the class data members directly.

Note: While data members in the C++ class can be **public**, **protected**, or **private**, the variables in the corresponding C structure are always publicly accessible. Be careful, you may eliminate the safeguards built into the C++ language.

You can use C++ operators on this class if you supply your own definitions of these operators in the form of member functions.

When you write your own C++ classes that you want to access from other languages:

- Do not use static data members in your class, because they are not part of the C++ object that is passed to the other language.
- Do not use virtual functions in your class, because you cannot access the data members since the alignment of the data members between the class and the C structure is different.
- By making all data members of a class publicly accessible to programs written in other languages, you may be breaking data encapsulation.

Using C++ Objects in a C Program



This program shows how you can access the data members in C++ classes from source code written in C.

Program Structure

The program consists of these files:

- A C++ source file `hourclas.cpp` which contains:
 - Definitions of one base class, `HourMin`, and two derived classes, `HourMinSec1` and `HourMinSec2`
 - Three function prototypes with extern "C" linkage:
 - extern "C" void `CSetHour(HourMin *)`
 - extern "C" void `CSetSec(HourMin *)`
 - extern "C" void `CSafeSetHour(HourMin *)`
 - The definition of a function with extern "C" linkage, extern "C" void `CXXSetHour(HourMin * x)`
 - A `main()` function containing the program logic
- A C source file `hour.c` which contains:
 - A structure `CHourMin` that maps to the C++ class `HourMin` in file `hourclas.cpp`
 - Definitions of the three functions with extern "C" linkage declared in `hourclas.cpp`
 - The definition of a function `CSafeSetHour()`

Program Description

In its `main()` function the program:

1. Instantiates an object `hm` of the base class `HourMin`
2. Assigns a value to the `h` variable (hour) in the base class
3. Passes the address of the base class to the function `CSetHour()` defined in the C source file `hour.c`, which assigns a new value to `h` in the base class
4. Displays the value of `h` in the base class
5. Instantiates an object `hms1` of the derived class `HourMinSec1`
6. Passes the address of this object class to the function `CSetSec()` defined in the C source file `hour.c`, which assigns a value to `s` in the object
7. Displays the value of `s` in the object
8. Instantiates an object `hms2` of the class `HourMinSec2` which contains the class `HourMin`
9. Passes the address of this new object to the function `CSetSec()` defined in the C source file `hour.c`, which assigns a value to `s` in the object
10. Displays the value of `s` in the object

11. Passes the address of the base class object to function SafeSetHour() defined in the C source file hour() which passes the address back to a function CXXSetHour() defined in the C++ source file hourclas.cpp

The C++ source file hourclas.cpp is:

```
#include <iostream.h>

class HourMin {          // base class
protected:
    int h;
    int m;

public:
    void set_hour(int hour) { h = hour % 24; } // keep it in range
    int  get_hour()         { return h; }
    void set_min(int min)   { m = min % 60; } // keep it in range
    int  get_min()         { return m; }
    HourMin(): h(0), m(0) {}
    void display() { cout << h << ':' << m << endl; }
};

// derived from class HourMin
class HourMinSec1 : public HourMin {
public:
    int s;
    void set_sec(int sec) { s = sec % 60; } // keep it in range
    int  get_sec()         { return s; }
    HourMinSec1() { s = 0; }
    void display() { cout << h << ':' << m << ':' << s << endl; }
};

// has an HourMin contained inside
class HourMinSec2 {
private:
    HourMin a;
    int s;

public:
    void set_sec(int sec) { s = sec % 60; } // keep it in range
    int  get_sec()         { return s; }
    HourMinSec2() { s = 0; }
    void display() {
        cout << a.get_hour() << ':' << a.get_min() << ':' << s << endl; }
};

extern "C" void CSetHour(HourMin *); // defined in C
extern "C" void CSetSec(HourMin *);  // defined in C
extern "C" void CSafeSetHour(HourMin *); // defined in C

// wrapper function to be called from C code */
extern "C" void CXXSetHour(HourMin * x) {
    x->set_hour(99); // much like the C version but the C++
                    // member functions provide some protection
                    // expect 99 % 24, or 3 to be the result
}

// other wrappers may be written to access other member functions
// or operators ...

main() {
    HourMin hm;
    hm.set_hour(18); // supper time;
    CSetHour(&hm);   // pass address of object to C function
    hm.display();    // hour is out of range
}
```

```

HourMinSec1 hms1;
CSetSec((HourMin *) &hms1)
hms1.display();

HourMinSec2 hms2;
CSetSec(&hms2);
hms2.display();

CSafeSetHour(&hm); // pass address to a safer C function
hm.display();    // hour is not out of range
}

```

The C source file `hour.c` is:

```

/* C code  hour.c */

struct CHourMin {
    int hour;
    int min;
};

void CSetHour(void * v) {
    struct CHourMin * p;
    p = (struct CHourMin *) v; // force it to the type we want
    p->hour = 99;              // with power comes responsibility (oops!)
}

struct CHourMinSec {
    struct CHourMin hourMin;
    int sec;
};

// handles both HourMinSec1, and HourMinSec2 classes

void CSetSec(void *v) {
    struct CHourMinSec * p;
    p = (struct CHourMinSec *) v; // force it to the type we want
    p->sec = 45;
}

void CSafeSetHour(void *v) {
    struct CHourMin * p;
    p = (struct CHourMin *) v; // force it to the type we want

    // ... do things with p, but be careful
    // ...
    // use a C++ wrapper function to access C++ function members

    CXXSetHour(p);    // almost the same as p->hour = 99
}

```

Program Output

The program output is:

```

99:0
0 -:0 :45
0 -:0 :45
3 -:0
Press ENTER to end terminal session.

```

Qualifying Library Calls

You can call a program with a library qualification by using the:

- Bindable APIs with library qualification
- `system()` function in C to use QCAPEXC and call with library qualification
- iSeries Resolve System Pointer (RSLSYIP) to resolve to the object with a library and call with the pointer

Calling OPM Programs

This program demonstrates some typical steps in creating a program that uses several ILE and OPM programming languages.

Program Description

The program is a small transaction-processing program that takes as input the item name, price, and quantity for one or more products. As output, the program displays the total cost of the items specified on the display and writes an audit trail of the transactions to a file.

Figure 198 shows the basic flow of the program.

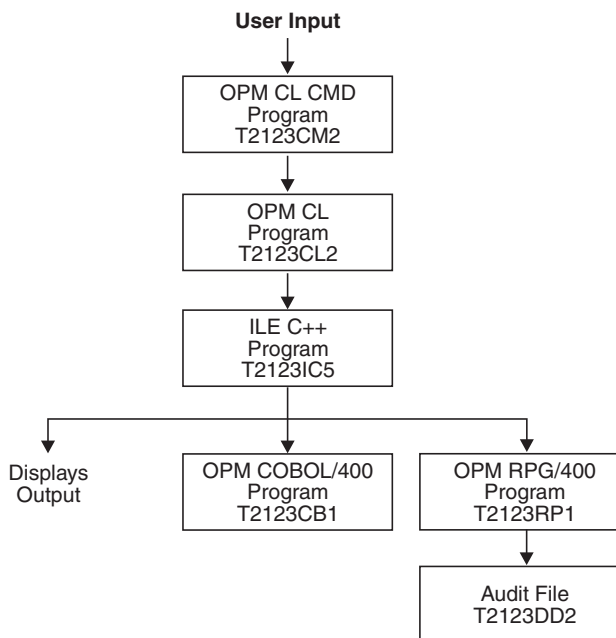


Figure 198. Basic Program Structure

Program Structure

The program consists of these components:

- A CL command T2123CM2 that accepts the users input and passes it to an OPM CL program
- An OPM CL program T2123CL2 that processes the input and passes it to an ILE C++ program
- An ILE C++ program T2123IC5 that calls an OPM COBOL program to process the input, and an OPM RPG program to write the audit trail to an externally described file

- An OPM COBOL program T2123CB1 that completes the calculation and formats the cost
- An OPM RPG program T2123RP1 that updates the audit file with each transaction
- An externally described file T2123DD2 that receives the audit trail

Program Activation

The ILE C++ program T2123IC5 is created with the CRTPGM default for the *ACTGRP* parameter, *ACTGRP(*NEW)*. When the CL program calls the ILE C++ program, a new activation group is started.

The OPM CL, COBOL, and RPG programs are activated within the OPM default activation group.

Figure 199 shows the structure of this program in ILE.

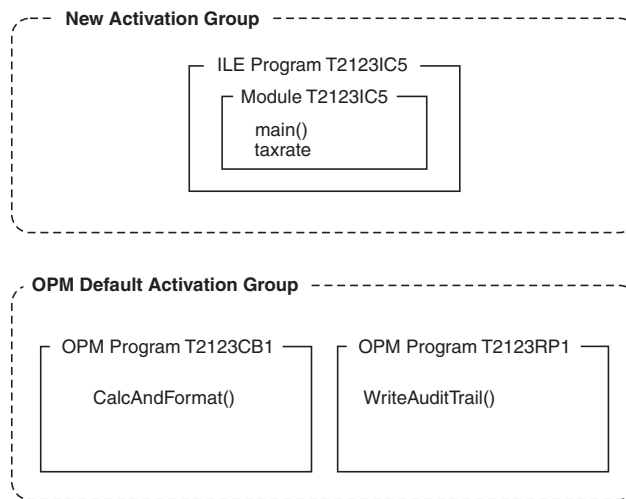


Figure 199. Structure of the Program in ILE C++

Program Files

The source code for each of the files that compose this program are an externally described file, a CL program, a CL command prompt, a C++ source file, and OPM COBOL program and an OPM RPG program.

Externally Described File T2123DD2

The file T2123DD2 contains the audit trail for the C++ program T2123IC5. The DDS source defines the fields for the audit file:

```

R T2123DD2R
  USER          10          COLHDG('User')
  ITEM          20          COLHDG('Item name')
  PRICE        10S 2        COLHDG('Unit price')
  QTY           4S          COLHDG('Number of items')
  TXRATE        2S 2        COLHDG('Current tax rate')
  TOTAL         21          COLHDG('Total cost')
  DATE          6          COLHDG('Transaction date')
K USER
  
```

CL Program T2123CL2

The CL program T2123CL2 passes the CL variables `item_name`, `price`, `quantity` and `user_id` by reference to an ILE C++ program T2123IC5.

```

PGM          PARM(&ITEMIN &PRICE &QUANTITY)
DCL          VAR(&USER) TYPE(*CHAR) LEN(10)
DCL          VAR(&USEROUT) TYPE(*CHAR) LEN(11)
DCL          VAR(&ITEMIN) TYPE(*CHAR) LEN(20)
DCL          VAR(&ITEMOUT) TYPE(*CHAR) LEN(21)
DCL          VAR(&PRICE) TYPE(*DEC) LEN(10 2)
DCL          VAR(&QUANTITY) TYPE(*DEC) LEN(2 0)
DCL          VAR(&NULL) TYPE(*CHAR) LEN(1) VALUE(X'00')
/* ADD NULL TERMINATOR FOR THE ILE C PROGRAM */
CHGVAR       VAR(&ITEMOUT) VALUE(&ITEMIN *TCAT &NULL)
CHGVAR       VAR(&USEROUT) VALUE(&USER *TCAT &NULL)
/* GET THE USERID FOR THE AUDIT FILE */
RTVJOBA      USER(&USER)
/* ENSURE AUDIT RECORDS WRITTEN TO CORRECT AUDIT FILE MEMBER */
OVRDBF       FILE(T2123DD2) TOFILE(*LIBL/T2123DD2) +
              MBR(T2123DD2) OVRSCOPE(*CALLLVL) SHARE(*NO)
CALL         PGM(T2123IC5) PARM(&ITEMOUT &PRICE &QUANTITY +
              &USEROUT)
DLTOVR       FILE(*ALL)
ENDPGM

```

The Retrieve Job Attributes (RTVJOBA) command obtains the user ID for the audit trail. Arguments are passed by reference. They can be changed by the receiving ILE C++ program. The variables containing the user and item names are explicitly null-terminated in the CL program.

Note: CL variables and numeric literals are not passed to an ILE C++ program with null-terminated strings. Character literals and logical literals are passed as null-terminated strings but are not widened with blanks. Numeric literals such as packed decimals are passed as 15,5 (8 bytes). Floating point constants are passed as double precision floating point values (1.2E+15).

CL Command Prompt T2123CM2

You use the CL command prompt T2123CM2 to prompt the user to enter item names, prices, and quantities that will be used by the C++ program T2123IC5.

```

CMD          PROMPT('CALCULATE TOTAL COST')
PARM         KWD(ITEM) TYPE(*CHAR) LEN(20) RSTD(*NO) +
              MIN(1) ALWUNPRT(*NO) PROMPT('Item name' 1)
PARM         KWD(PRICE) TYPE(*DEC) LEN(10 2) RSTD(*NO) +
              RANGE(0.01 99999999.99) MIN(1) +
              ALWUNPRT(*YES) PROMPT('Unit price' 2)
PARM         KWD(QUANTITY) TYPE(*INT2) RSTD(*NO) RANGE(1 +
              9999) MIN(1) ALWUNPRT(*YES) +
              PROMPT('Number of items' 3)

```

C++ Source File T2123IC5

The C++ source file T2123IC5 contains a main() function which receives the incoming arguments from the CL program T2123CL2. These arguments have been verified by the CL command prompt T2123CM2 and null-terminated within the CL program T2123CL2. All the incoming arguments are pointers.

The main() function calls the function CalcAndFormat() which is mapped to a COBOL name. It passes the price, quantity, taxrate, formatted_cost, and a success_flag to the OPM COBOL program T2123CB1 using the extern "OS nowiden" linkage specification, because the OPM COBOL program is not expecting widened parameters.

The formatted_cost and the success_flag values are updated in the C++ program T2123IC5.

If `CalcAndFormat()` returns successfully a record is written to the audit trail by `WriteAuditTrail()` in the OPM RPG program.

The `main()` function in program `T2123IC5` calls `WriteAuditTrail()` which is mapped to an RPG program name, and passes the `user_id`, `item_name`, `price`, `quantity`, `taxrate`, and `formatted_cost`, using the extern "OS" linkage specification.

Note: By default, the compiler converts a short integer to an integer unless the `nowiden` parameter is specified on the extern linkage specification. The short integer in the C++ program is converted to an integer, and then passed to the OPM RPG program. The RPG program is expecting a 4 byte integer for the `quantity` variable. See "Understanding Data-Type Compatibility" on page 361 for information on data-type compatibility.

```
// This program is called by a CL program that passes an item
// name, price, quantity and user ID.
// COBOL is called to calculate and format the total cost.
// RPG is called to write an audit trail.

#include <iostream.h>
#include <stdlib.h>
#include <string.h>
#include <bcd.h>

// The #pragma map directive maps a new program name to the existing
// program name so that the purpose of the program is clear.
// Tell the compiler that there are dynamic program calls so
// arguments are passed by value-reference.

extern "OS nowiden" void CalcAndFormat(_DecimalT <10,2>,
                                     short int, _DecimalT<2,2>, char[],
                                     char *);

#pragma map(CalcAndFormat,"T2123CB1")

extern "OS" void WriteAuditTrail(char[], char[],
                                _DecimalT<10,2>, short int,
                                _DecimalT<2,2>, char[]);

#pragma map(WriteAuditTrail,"T2123RP1")

int main(int argc, char *argv[])
{
    // Incoming arguments from a CL program have been verified by
    // the *CMD and null-terminated within the CL program.
    // Incoming arguments are passed by reference from a CL program.

    char          *user_id;
    char          *item_name;
    short int     quantity;
    _DecimalT <10, 2> price;
    _DecimalT <2,2> taxrate = __D(".15");
    char          formatted_cost[22];

    // Remove null terminator for RPG program. Item name is null
    // terminated for C++.

    char          rpg_item_name[20];
    char          null_formatted_cost[22];
    char          success_flag = 'N';
    int           i;

    // Incoming arguments are all pointers.
```

```

    item_name = argv[1];
    price      = *((_DecimalT<10, 2> *) argv[2]);
    quantity   = *((short *) argv[3]);
    user_id    = argv[4];

// Call the COBOL program to do the calculation, and return a
// Y/N flag, and a formatted result.

    CalcAndFormat(price, quantity, taxrate, formatted_cost,
                  &success_flag);

    memcpy(null_formatted_cost, formatted_cost, sizeof(formatted_cost));

// Null terminate the result.

    formatted_cost[21] = '\0';
    if (success_flag == 'Y')
    {
        for (i=0; i<20; i++)
        {

// Remove null terminator for the RPG program.

            if (*(item_name+i) == '\0')
            {
                rpg_item_name[i] = ' ';
            }
            else
            {
                rpg_item_name[i] = *(item_name+i);
            }
        }

// Call an RPG program to write audit records.

        WriteAuditTrail(user_id, rpg_item_name, price, quantity,
                        taxrate, formatted_cost);

        cout <<quantity <<item_name << "plus tax ="
              <<null_formatted_cost <<endl;
    }
    else
    {
        cout <<"Calculation failed" <<endl;
    }
}

```

OPM COBOL Program T2123CB1

The OPM COBOL program T2123CB1 receives pointers to the values of the variables price, quantity and taxrate, and pointers to formatted_cost and success_flag.

The CalcAndFormat() function in program T2123CB1 calculates and formats the total cost. Parameters are passed from the ILE C++ program to the OPM COBOL program to do the tax calculation.

```

IDENTIFICATION DIVISION.
PROGRAM-ID. T2123CB1.
*****
* parameters:                                     *
*   incoming:  PRICE, QUANTITY                   *
*   returns :  TOTAL-COST (PRICE*QUANTITY*1.TAXRATE) *
*              SUCCESS-FLAG.                     *
*****
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.

```

0
0


```

SOURCE-COMPUTER. AS-400.                                0
OBJECT-COMPUTER. AS-400.                                0
DATA DIVISION.                                           0
WORKING-STORAGE SECTION.

    01 WS-TOTAL-COST          PIC S9(13)V99              COMP-3.
    01 WS-TAXRATE             PIC S9V99                  COMP-3.

LINKAGE SECTION.

    01 LS-PRICE                PIC S9(8)V9(2)            COMP-3.
    01 LS-QUANTITY             PIC S9(4)                  COMP-4.
    01 LS-TAXRATE              PIC S9V99                  COMP-3.
    01 LS-TOTAL-COST           PIC $$$,$$$,$$$,$$$,$$.99
                                DISPLAY.
    01 LS-OPERATION-SUCCESSFUL PIC X                     DISPLAY.

PROCEDURE DIVISION USING LS-PRICE                        0
                        LS-QUANTITY
                        LS-TAXRATE
                        LS-TOTAL-COST
                        LS-OPERATION-SUCCESSFUL.

MAINLINE.
    MOVE "Y" TO LS-OPERATION-SUCCESSFUL.
    PERFORM CALCULATE-COST.
    PERFORM FORMAT-COST.
    EXIT PROGRAM.

CALCULATE-COST.
    MOVE LS-TAXRATE TO WS-TAXRATE.
    ADD 1 TO WS-TAXRATE.
    COMPUTE WS-TOTAL-COST ROUNDED = LS-QUANTITY *
                                LS-PRICE *
                                WS-TAXRATE

    ON SIZE ERROR
        MOVE "N" TO LS-OPERATION-SUCCESSFUL
    END-COMPUTE.

FORMAT-COST.
    MOVE WS-TOTAL-COST TO LS-TOTAL-COST.

```

OPM RPG Program T2123RP1

The OPM RPG program T2123RP1 contains the WriteAuditTrail() function which writes the audit trail for the program.

```

FT2123DD20  E          DISK          A
F           T2123DD2R          KRENAMEDD2R
IQTYIN      DS
I
C           *ENTRY   PLIST
C           PARM     USER    10
C           PARM     ITEM    20
C           PARM     PRICE   102
C           PARM     QTYIN
C           PARM     TXRATE   22
C           PARM     TOTAL   21
C           EXSR ADDREC
C           SETON
C           ADDRREC  BEGSR
C           MOVE LUDATE  DATE
C           MOVE QTYBIN  QTY
C           WRITEDD2R
C           ENDSR

```

Invoking the ILE-OPM Program

To enter data for the program T2123IC5 enter the command T2123CM2 and press F4 (Prompt).

You can enter this data into T2123CM2:

```
Hammers
1.98
5000
Nails
0.25
2000
```

The output is:

```
5000 HAMMERS plus tax =          $11,385.00
Press ENTER to end terminal session.
>
2000 NAILS plus tax =           $575.00
Press ENTER to end terminal session.
```

The physical file T2123DD2 contains this data:

SMITHE	HAMMERS	00000000198500015	\$11,385.0007	2893
SMITHE	NAILS	00000000025200015	\$575.0007	2893

Calling ILE Programs

This program shows you some typical steps in creating a program that uses several ILE programming languages.

Program Description

This program is an ILE version of the small transaction-processing program described in the “Calling OPM Programs” on page 376.

Program Structure

The program consists of these components:

- A CL command T2123CM3 that accepts the user input and passes it to an ILE CL program
- An ILE CL program T2123CL3 that processes the input and passes it to an ILE program
- An ILE program T2123ICB in which the `main()` function of a C++ module T2123ICB calls a procedure `CalcAndFormat` in an ILE COBOL module T2123CB2
- A service program T2123SP3, created from a C++ source file `t2123icc.cpp`, that exports the variable `TAXRATE`
- A service program T2123SP4, created from an ILE RPG module object T2123RP2, that writes an audit trail of all transactions to a file
- An externally described file T2123DD2 that receives the audit trail data

Figure 200 on page 383 shows the ILE structure.

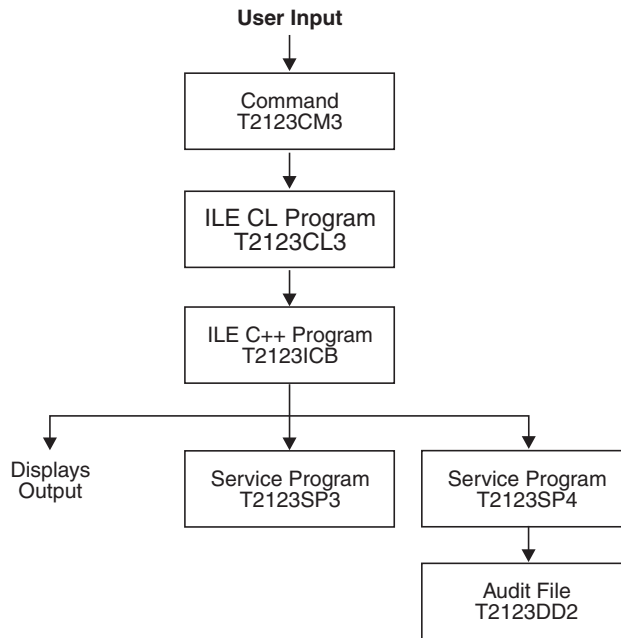


Figure 200. ILE Structure

Program Activation

The programs T2123CL3 and T2123ICB are created with the CRTPGM default for the *ACTGRP* parameter, *ACTGRP(*NEW)*. When the CL program calls the ILE C++ program, a new activation group is started.

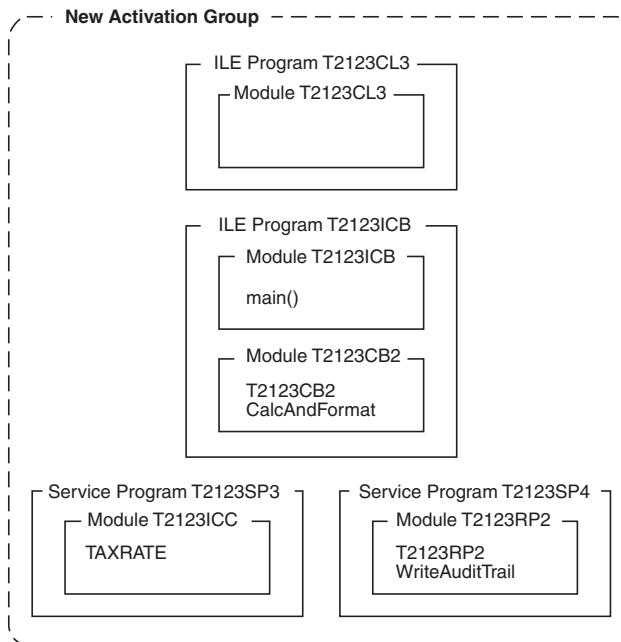


Figure 201. Basic Object Structure

The service programs are created with the CRTSRVPGM default for the *ACTGRP* parameter, *ACTGRP(*CALLER)*. When they are called, they are activated within the activation group of the calling program.

Figure 201 on page 383 shows the basic object structure used in this example.

Program Files

The source files for this program include an externally described file, a CL program, a command prompt, two C++ source files, and ILE COBOL source file and an ILE RPG source file.

Externally Described File T2123DD2

The file T2123DD2 contains the audit trail for the C++ program T2123ICB. The DDS source defines the fields for the audit file.

See “Externally Described File T2123DD2” on page 377 for the DDS source of the audit file T2123DD2.

CL Program T2123CL3

The CL program T2123CL3 passes the CL variables *item_name*, *price*, *quantity*, and *user_id* by reference to an ILE C++ program T2123IC5.

CL Command Prompt T2123CM3

You use the CL command prompt T2123CM3 to prompt the user to enter item names, prices, and quantities that will be used by the C++ program T2123ICB.

The source code for program T2123CL3 is identical to the source code shown in “CL Program T2123CL2” on page 377.

C++ Source File T2123ICB.CPP

The source for the ILE C++ program T2123ICB is almost identical to the source shown in “C++ Source File T2123IC5” on page 378. The difference lies in the linkage specifications used for interlanguage calls:

```
// This program demonstrates the interlanguage call capability
// of an ILE C++ program. This program is called by a CL
// program that passes an item name, price, quantity and user ID.
// A COBOL procedure is called to calculate and format total
// cost. An RPG procedure is called to write an audit trail.

#include <iostream.h>
#include <stdlib.h>
#include <string.h>
#include <bcd.h>

// The #pragma map directive maps a function name to the bound
// procedure name so that the purpose of the procedure is clear.
// Tell the compiler that there are bound procedure calls and
// arguments are to be passed by value-reference.

extern "COBOL" void CalcAndFormat(_DecimalT <10,2>,
                                short int, char[],
                                char *);

#pragma map(CalcAndFormat,"T2123CB2")

extern "RPG" void WriteAuditTrail(char[],
                                char[],
                                _DecimalT<10,2>,
                                short int, char[]);

#pragma map(WriteAuditTrail,"T2123RP2")
```

```

int main(int argc, char *argv[])
{
    // Incoming arguments from a CL program have been verified by
    // the *CMD and null-terminated within the CL program.
    // Incoming arguments are passed by reference from a CL program.

    char          *user_id;
    char          *item_name;
    short int     quantity;
    _DecimalT<10, 2> price;
    char          formatted_cost[22];

    // Remove null terminator for RPG program. Item name is null
    // terminated for C++.

    char          rpg_item_name[20];
    char          null_formatted_cost[22];
    char          success_flag = 'N';
    int           i;

    //Incoming arguments are all pointers.
    item_name = argv[1];
    price = *((_DecimalT<10, 2> *) argv[2]);
    quantity = *((short *) argv[3]);
    user_id = argv[4];

    // Call the COBOL program to do the calculation, and return a
    // Y/N flag, and a formatted result.

    CalcAndFormat(price, quantity, formatted_cost, &success_flag);

    memcpy(null_formatted_cost, formatted_cost, sizeof(formatted_cost));

    // Null terminate the result.

    formatted_cost[21] = '\0';
    if (success_flag == 'Y')
    {
        for (i=0; i<20; i++)
        {

            // Remove null terminator for the RPG program.

            if (*(item_name+i) == '\0')
            {
                rpg_item_name[i] = ' ';
            }
            else
            {
                rpg_item_name[i] = *(item_name+i);
            }
        }
    }

    // Call an RPG program to write audit records.

    WriteAuditTrail(user_id, rpg_item_name, price, quantity,
                    formatted_cost);

    cout <<"plus tax =" << quantity << item_name << null_formatted_cost
        <<endl <<endl;
    }
    else
    {
        cout <<"Calculation failed" <<endl;
    }
}

```

C++ Source File T2123ICC

The source for the ILE C++ module T2123ICC shows the variable TAXRATE is exported from this module to be used by ILE COBOL and ILE RPG procedures.

Note: Weak definitions (EXTERNALs from COBOL) cannot be exported out of a service program to a strong definition language like C or C++, while C or C++ can export to COBOL. The choice of language for TAXRATE is C++.

```
// Export the tax rate data.
#include <bcd.h>
const _DecimalT <2,2> TAXRATE = __D(".15");
```

ILE COBOL Module T2123CB2

The ILE COBOL procedure in T2123CB2 receives pointers to the values of the variables price, quantity and taxrate, and pointers to formatted_cost and success_flag.

The CalcAndFormat() function calculates and formats the total cost. Parameters are passed from the ILE C++ program to the ILE COBOL procedure to do the tax calculation.

The source code for T2123CB2 is:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. T1520CB2 INITIAL.
*****
* parameters:
* incoming:  PRICE, QUANTITY
* returns :  TOTAL-COST (PRICE*QUANTITY*1.TAXRATE) *
*           SUCCESS-FLAG.
* TAXRATE :  An imported value.
*****
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. AS-400.
OBJECT-COMPUTER. AS-400.
DATA DIVISION.
WORKING-STORAGE SECTION.

    01 WS-TOTAL-COST          PIC S9(13)V99      COMP-3.
    01 WS-TAXRATE             PIC S9V99          COMP-3
                                     VALUE 1.
    01 TAXRATE                 EXTERNAL          PIC SV99      COMP-3.

LINKAGE SECTION.
    01 LS-PRICE                PIC S9(8)V9(2)     COMP-3.
    01 LS-QUANTITY              PIC S9(4)          COMP-4.
    01 LS-TOTAL-COST            PIC $$$,$$$,$$$,$$$,$$.99
                                     DISPLAY.
    01 LS-OPERATION-SUCCESSFUL PIC X              DISPLAY.

PROCEDURE DIVISION USING LS-PRICE
                       LS-QUANTITY
                       LS-TOTAL-COST
                       LS-OPERATION-SUCCESSFUL.

MAINLINE.
    MOVE "Y" TO LS-OPERATION-SUCCESSFUL.
    PERFORM CALCULATE-COST.
    PERFORM FORMAT-COST.
    EXIT PROGRAM.

CALCULATE-COST.
    ADD TAXRATE TO WS-TAXRATE.
    COMPUTE WS-TOTAL-COST ROUNDED = LS-QUANTITY *
```

```

LS-PRICE *
WS-TAXRATE

ON SIZE ERROR
  MOVE "N" TO LS-OPERATION-SUCCESSFUL
END-COMPUTE.

FORMAT-COST.
  MOVE WS-TOTAL-COST TO LS-TOTAL-COST.

```

ILE RPG Module T2123RP2

The ILE RPG module T2123RP2 contains the WriteAuditTrail() function which writes the audit trail for the program:

```

FT1520DD2  0  A  E          DISK
D  TAXRATE          S          3P 2  IMPORT
D  QTYIN            DS
D  QTYBIN          1      4B  0
C    *ENTRY          PLIST
C                      PARM          USER          10
C                      PARM          ITEM          20
C                      PARM          PRICE         10 2
C                      PARM          QTYIN
C                      PARM          TOTAL          21
C                      EXSR          ADDREC
C                      SETON
C    ADDREC          BEGSR
C                      MOVE          UDATE          DATE
C                      MOVE          QTYBIN          QTY
C                      MOVE          TAXRATE         TXRATE
C                      WRITE          T1520DD2R
C                      ENDSR

```

Service Program T2123SP3

Service program T2123SP3 is created from the C++ module T2123ICC. It exports the variable TAXRATE.

Service Program T2123SP4

Service program T2123SP4 is created from the ILE RPG module T2123RP2. It exports the procedure T2123RP2.

Invoking the ILE Program

T2123ICB is considered the main program. It runs in the new activation group that is created when the CL program T2123CL3 is called.

To enter data for the program T2123ICB enter the command: T2123CM2 and press F4 (Prompt). You can enter the sample data in “Invoking the ILE-OPM Program” on page 382.

The output is the same as for the OPM version of this program.

The physical file T2123DD2 contains the same data as shown in the OPM version in “Invoking the ILE-OPM Program” on page 382.

Calling an ILE C++ Program

This program shows how to retrieve a return value from main. A CL command called SQUARE calls an ILE C++ program SQITF. The program SQITF calls another ILE C++ program called SQ. The program SQ returns a value to program SQITF.

Note: Returning an integer value from an ILE C++ program may impact performance.

You use the CL command prompt SQUARE to enter the number you want to determine the square of for the ILE C++ program SQITF:

```
CMD      PROMPT('CALCULATE THE SQUARE')
PARM     KWD(VALUE) TYPE(*INT4) RSTD(*NO) RANGE(1 +
          9999) MIN(1) ALWUNPRT(*YES) PROMPT('Value' 1)
```

The ILE C++ program calls another ILE C++ program called SQ:

```
// This program SQITF is called by the command SQUARE. This
// program then calls another ILE C++ program SQ to perform
// calculations and return a value.

#include <iostream.h>

extern "OS" int SQ(int); // Tell compiler this is external call,
                        // do not pass by value.

int main(int argc, char *argv[])
{
    int *x;
    int result;

    x = (int *) argv[1];

    result = SQ(*x);

    // Note that although the argument is passed by value, the compiler
    // copies the argument to a temporary variable, and the pointer to
    // the temporary variable is passed to the called program SQ.

    cout <<"The SQUARE of" <<x <<"is" <<result <<endl;
}
```

The ILE C++ program SQ calculates an integer value and returns the value to the calling program SQITF:

```
// This program is called by a ILE C++ program called SQITF.
// It performs the square calculations and returns a value to SQITF.

int main(int argc, char *argv[])

{ return (*(int *) argv[1]) * (*(int *) argv[1]);
}
```

To enter data for the program SQITF enter the command SQUARE and press F4 (Prompt). Type 10, and press Enter. The output is:

```
The SQUARE of 10 is 100
Press ENTER to end terminal session.
```

Calling an EPM C Program

If your ILE C++ program calls an EPM default entry point, use the extern linkage specification in your ILE C++ source to tell the compiler that PGMNAME is an external program, not a bound ILE procedure. QPXXCALL is not needed to call EPM default entry points.

The ILE C++ program T2123DL2 passes two integers and two characters to an EPM C program T2123DL3:

```
// t2123dl2.cpp

#include <string.h>
#include <iostream.h>
```



```
extern "OS" void T2123DL3(int *,int *,char *,char *);

int main(void)
{
    int      i;          // Integer parameter to pass to callee.
    int      rtn;        // Return value from main in T2123DL3.
    char      a,b;       // Character parameters to pass.
    i = 5;              // Initialize parameters to be passed.
    a = 'a';
    b = 'b';

    T2123DL3(&rtn,&i,&a,&b);

    cout << "Values returned are rtn = " <<rtn << ", i=" <<i
        << ", a=" <<a << ", b=" <<b <<endl;
}
```

The EPM C program T2123DL3 receives and return values to the ILE C++ program:

```
/* This program illustrates how this EPM C program retrieves */
/* t2123dl3.c */

#include <stdio.h>

int main( int argc, char *argv[])
{
    printf ( "integer passed was %d\n", * (int *) argv[2]);

        * (int *) argv[2] = 8;
    printf ( "character passed was %c\n", *argv[3]);
        *argv[3] = 'D';

    printf ( "integer passed was %d\n", * (int *) argv[4]);

    printf ( "as a character it is %c\n", *argv[4]);

    printf ( "returning %d\n", * (int *) argv[1]);
    * (int *) argv[1] =5;

}
```

The output is:

```
Values returned are rtn = 5, i = 8 a = D b = b
Press ENTER to end terminal session.
```

```
Start of terminal session.
integer passed was 5
character passed was a
integer passed was -2113929216
as a character it is b
returning 0
Press ENTER to end terminal session.
```

Calling ILE-Bindable APIs

The program T2123API uses DSM ILE bindable API calls to create a window and echo whatever is entered. The *System API Reference* contains information on the ILE bindable APIs. The prototypes for the DSM APIs are in the <qsnssess.h> header file. The extern "OS nowiden" return type API(arg_list); is specified for each

API where return type is void or whatever type is returned by the API, and arg_list is the list of parameters taken by the API. This ensures any value argument is passed by value indirectly.

```
// This program uses Dynamic Screen Manager API calls to
// create a window and echo whatever is entered. This is an
// example of bound API calls. Note the use of extern linkage
// in the <qsnssess.h> header file. OS, nowiden ensures that a
// pointer to an unwidened copy of the argument is passed to the
// API.
// Use BNDDIR(QSNAPI) on the CRTPGM command to build this
// example.

#include <stddef.h>
#include <string.h>
#include <iostream.h>
#include <qsnapi.h>

#define BOTLINE " Echo lines until:  PF3 - exit"

// DSM Session Descriptor Structure.

typedef struct{
    Qsn_Ssn_Desc_T sess_desc;
    char          buffer[300];
}storage_t;

void F3Exit(const Qsn_Ssn_T *Ssn, const Qsn_Inp_Buf_T *Buf, char *action)
{
    *action = '1';
}

int main(void)
{
    int i;
    storage_t    storage;

    // Declarators for declaring windows. Types are from the <qsnssess.h>
    // header file.

    Qsn_Inp_Buf_T    input_buffer = 0;
    Q_Bin4           input_buffer_size = 50;
    char             char_buffer[100];
    Q_Bin4           char_buffer_size;

    Qsn_Ssn_T        session1;
    Qsn_Ssn_Desc_T   *sess_desc = (Qsn_Ssn_Desc_T *) &storage;
    Qsn_Win_Desc_T   win_desc;
    Q_Bin4           win_desc_length = sizeof(Qsn_Win_Desc_T);
    char             *botline = BOTLINE;
    Q_Bin4           botline_len = sizeof(BOTLINE) - 1;
    Q_Bin4           sess_desc_length = sizeof(Qsn_Ssn_Desc_T) +
                                     botline_len;

    Q_Bin4           bytes_read;

    // Initialize Session Descriptor DSM API.

    QsnInzSsnD( sess_desc, sess_desc_length, NULL);

    // Initialize Window Descriptor DSM API.

    QsnInzWinD( &win_desc, win_desc_length, NULL);

    sess_desc->cmd_key_desc_line_1_offset = sizeof(Qsn_Ssn_Desc_T);
    sess_desc->cmd_key_desc_line_1_len = botline_len;
    memcpy( storage.buffer, botline, botline_len );
}
```

Passing Operational Descriptors

This program shows you how to use operational descriptors in ILE C++. It shows:

- An operational descriptor for func1 with a **#pragma descriptor** directive for the function in a header file "op_desc.h"
- An ILE C++ program T2123CP1 that calls func1 in a C++ module t2123CP2
- The ILE C++ source code of module T2123CP2 that contains an ILE API that is used to get information from the operational descriptor

The source code for the header file op_desc.h shows an operational descriptor for func1 with a **#pragma descriptor** directive for the function:

```
/* op_desc.h */
/* containing function prototype */
extern "C" int func1( char a[5], char b[5],
char *c );
#pragma descriptor(void func1( "", "", "" ))
```

A function func1() is declared. The **#pragma descriptor** directive for func1() specifies that the compiler must generate string operational descriptors for the three arguments the function takes.

The source code t2123cp1.cpp shows that the program T2123CP1 calls function func1(). When the function is called, the compiler generates operational descriptors for the three arguments specified on the call:

```
// t2123cp1.cpp

#include "op_desc.h"

main()
{
    char a[5] = {'s', 't', 'u', 'v', '\0'};
    char *c;

    c = "EFGH";
    func1(a, "ABCD", c);
}
```

The source code t2123cp2.cpp for module T2123CP2 defines function func1(). It contains the call to the ILE API that is used to determine the string type, length, and maximum length of the string arguments declared in function func1().

The *System API Programming* contains information about _FEEDBACK. The values for typeCharZ and typeCharV2 are found in the ILE API header file <leod.h>.

```
// t2123cp2.cpp

#include <string.h>
#include <iostream.h>
#include <leawi.h>
#include <leod.h>
#include "op_desc.h"

int func1(char a[5], char b[5], char *c)
{
    int      posn      = 1;
    int      datatype;
    int      currlen;
    int      maxlen;
    _FEEDBACK fc;

    char      *string1;
```

```

/* Call to ILE API CEEGSI to determine string type, length*/
/* and the maximum length.*/

CEEGSI(&posn, &datatype, &curlen, &maxlen, &fc);

switch(datatype)
{
    case typeCharZ:
        string1 = a;
        break;
    case typeCharV2:
        string1 = a + 2;
        break;
}

/* Use string1.*/

if (!memcmp(string1, "stuv", curlen))
    cout <<"First 4 characters are the same."<<endl;
else
    cout <<"First 4 characters are not the same."<<endl;
return 0;
}

```

Chapter 17. Using Teraspace

Teraspace is an extension of the iSeries storage model and run-time environment. It assists porting of applications to the iSeries system, and supports the continuous evolution of existing iSeries and OS/400 applications.

Previously, only 16 megabytes (minus 4 kilobytes) of contiguous address range could be addressed. Teraspace supports much larger contiguous address ranges (up to 4 GB in V5R1). Programs can obtain working storage from teraspace and can also use 8-byte pointers to address teraspace.

The teraspace storage (TERASPACE), storage model (STGMDL), and data model (DTAMD) options of the C/C++ compilers determine the environment for teraspace. For more information about these options, see *ILE C/C++ Compiler Reference*.

Before trying to use teraspace in your C/C++ programs, see the chapter on *Teraspace and single-level store* in *ILE Concepts*.

Pointer Support in the C/C++ Compilers

Traditional 16-Byte Pointers

Traditional iSeries-specific pointers `_SPCPTR`, `_SYSPTR`, `_INVPTR`, `_LBLPTR`, `_SUSPENDPTR`, and `_OPENPTR` are tagged, 16 bytes in length, and 16-byte aligned, regardless of the data model. A variable declared to be of type `_OPENPTR` can hold a value which is of any iSeries tagged pointer type. A 16-byte pointer of type `void*` is an open pointer. 16-byte pointers can be used to access both teraspace and single-level storage.

8-Byte (Process Local) Pointers

8-byte pointers have no special alignment requirement (although 8-byte is best for performance). The value of an 8-byte pointer can be manipulated and created using arithmetic and logical operations. There is no process local pointer equivalent to the iSeries open pointer. An 8-byte pointer of type `void*` is not an open pointer. To declare an open pointer, the pointer variable must be qualified with `__ptr128`. 8-byte pointers can only be used to access teraspace storage of the current job.

Pointer Declarations

8-byte and 16-byte pointers can be declared in any of the following ways:

- Pointer modifiers `__ptr64` and `__ptr128`
- DTAMD compiler option
- `datamodel` pragma

Pointer Conversions

The C/C++ compilers perform conversions between 8-byte and 16-byte pointers via assignment and parameter association. No warnings are issued when these conversions occur.

- Compiler conversions only apply to the first level of pointer. For example, a `char**` interface in the P128 data model will cause problems for a program using the LLP64 data model, because the compiler will only convert the first or "outer" level of pointer. In this case, you must explicitly specify a `__ptr128` qualifier for the inner level of pointer.

C shorthand syntax for the address of an array can further complicate the issue. In the previous example, if a program tried to pass `&arrayName`, where `arrayName` is declared as `char[n]`, to the P128 `char**` interface, the programmer must explicitly declare a `__p128` pointer and assign `arrayName` to that pointer.

- An interface which specifies 16-byte pointers can be called from source code passing 8-byte pointers, or vice versa. Explicit users of pointer modifiers or type castings are not required in the calling code.
- Assignments from 8-byte pointers to 16-byte pointers are always valid.
- Assignments from 16-byte to 8-byte pointers are valid only when the 16-byte pointers point to teraspace storage. The built-in function `_RETTSADR` signals the exception MCH0609 when used to convert 16-byte pointers to 8-byte pointers.
- Prior to comparison to 16-byte pointers, 8-byte pointers are converted to 16-byte pointers.

Using Teraspace in Your C/C++ Programs

When using teraspace in your C/C++ programs, you should be aware of the following considerations:

Linkage to `main()`

The `argv[]` array of pointers passed into `main()` is consistent with the explicit declaration or data model in effect for `main()`. For example, if data model LLP64 is in effect, or an explicit `__ptr64` declaration is used for `argv` (that is, `*__ptr64 *__ptr64 argv`), and array of process local pointers is created by the C/C++ compilers.

The following declarations of `main()` are valid:

- `main(int argc, char * * argv)` when the data model is P128 or LLP64
- `main(int argc, char * argv[])` when the data model is P128 or LLP64
- `main(int argc, char *__ptr128 *__ptr128 argv)`
- `main(int argc, char *__ptr128 argv[])` when the data model is P128
- `main(int argc, char *__ptr64 *__ptr64 argv)`
- `main(int argc, char *__ptr64 argv[])` when the data model is LLP64

The C/C++ compilers issue an error for the following declarations of `main()` which consists of mixed pointer sizes for `argv`:

- `main(int argc, char *__ptr64 *__ptr128 argv)`
- `main(int argc, char *__ptr128 *__ptr64 argv)`
- `main(int argc, char *__ptr64 * argv)` when the data model is P128
- `main(int argc, char *__ptr64 argv[])` when the data model is P128
- `main(int argc, char *__ptr128 * argv)` when the data model is LLP64
- `main(int argc, char *__ptr128 argv[])` when the data model is LLP64

The same rule also applies to the third parameter, `envp`, in `main()`. Moreover, the resulting pointer sizes of `argv` and `envp` must also be the same.

Variable Argument Lists

A function declared with a variable argument list is governed by the data model at the site of declaration. The variable portion of an argument list is

always homogeneous in terms of the size of pointer variables. Conversions of pointers in the variable portion are supplied by the C/C++ compilers.

Unprototyped Functions

An unprototyped function has its signature inferred by the data model in effect at the time of its first reference. The C runtime functions may produce unpredictable results if the header files are not included.

Forward Declarations

A class or structure uses the data model in effect when that class or structure is fully declared. This data model may be different from the data model in effect when that same class or structure is forward-declared.

For example, in the code segment below, struct Foo does not use the P128 data model in effect at the time of its forward-declaration. Instead, struct Foo uses the LLP64 data model in effect at the time struct Foo is fully declared.

```
#pragma datamodel(P128)
struct Foo; //forward declaration
#pragma datamodel(LLP64)
struct Foo {
    char* string; // this pointer is 8-byte because LLP64 datamodel
                  // was in effect for the struct definition.
};
#pragma datamodel(pop)
#pragma datamodel(pop)
```

Arrays

Arrays are not pointers, so they are not affected by pointer modifiers. Using an array in pointer context is the same as taking the address of the first element in the array. The size of the address is 8-byte if the storage model is teraspace.

The Address of (&) Operator

The address of (&) operator returns an 8-byte result if the storage model is teraspace.

The New and Delete Operators

To create modules which allocate dynamically from teraspace, specify TERASPACE(*YES *TSIFC) on the CRTCPMOD and CRTBNDCPP compiler commands. This remaps C runtime functions (for example, malloc/calloc/free) to new teraspace equivalents. This does not affect storage dynamically allocated using C++'s new and delete operators. If you want the new and delete operators to allocate storage from teraspace, you can override these operators.

The following example code overrides the global new and delete operators to call malloc/free functions which deal with teraspace.

```
/*
 * C++ global operator new and delete
 */

#include <stdlib.h>
#include <new.h>

void *operator new(size_t size)
throw std::bad_alloc
{
    // if size == 0, we must return a valid object! (ARM 5.3.3)
    if (size <= 0)
        size = 1;
```

```

void *ret = (malloc)(size);

while (ret == NULL)
{
    // The malloc failed. Call the new_handler to try to free more memory
    void (*temp_new_handler)();

    // First check to see if a thread local new handler is defined.
    temp_new_handler = _set_mt_new_handler(0);
    _set_mt_new_handler(temp_new_handler);

    // If there is no thread local handler try the global handler.
    if (temp_new_handler == NULL)
    {
        // Note that the following code is not thread safe
        // If the application is threaded and a new handler is set then
        // all calls to set_new_handler() in the application must be
        // blocked. Otherwise for the sake of speed, do not use a locking
        // mechanism.

        temp_new_handler = set_new_handler(0);
        set_new_handler(temp_new_handler);
    }

    if (temp_new_handler != NULL)
    {
        temp_new_handler();

        // Try one more time
        ret = (malloc)(size);
    }
    else
        throw std::bad_alloc;
}

// just return the result to the user
return ret;
}

/*
 * C++ global operator delete
 */

void operator delete(void *ptr)
{
    // delete of NULL is okay
    if (ptr == NULL)
        return;

    (free)(ptr);
}

```

Reference Types

Reference type are 8 bytes in length when the data model is LLP64.
Reference types can not be conditioned by the **__ptr64** modifier.

"this" Pointer

The size of "this" pointer depends on the effective data model of the class.
The prototype for the constructors, destructors and member functions of the "this" pointer will be marked accordingly with either **__ptr64** or **__ptr128**.

Templates

The template adopts the data model in effect when it is declared, and applies that data model to future instantiations of the template. In the following example:

- Any instantiation of FooT uses the P128 data model
- Any instantiation of FooTZ uses the LLP64 data model

```
#pragma datamodel(LLP64)
template <class T>
class FooTZ {
    public:
        T bar(const char * a, T x) { return x; }
};
#pragma datamodel(pop)

#pragma datamodel(P128)
template <class T>
class FooT {
    public:
        T bar(const char * a, T x) {return x; }
};
#pragma datamodel (pop)
```

Function Overloading

A function signature is affected by the data model governing the class/function declaration. For example, `int Bar::foo(const char *)` is mangled to:

- `foo__3BarFPCc` when the data model is P128
- `foo__3BarZFPCc` when the data model is LLP64

It is possible to create overloaded methods which are identical in every way except the size of a pointer argument. For example:

```
class Bar {
    int foo(const char __ptr128);
    int foo(const char __ptr64);
};
```

Internal Data Members

To be compatible with the current common runtime, all internal structures (for example, virtual function tables) use 16-byte pointers.

Class Inheritance

A derived class must be of the same data model as the base class.

Part 7. Using International Locales and Coded Character Sets

This part describes how to:

- Work with alternate coded character sets
- Use international locales

Chapter 18. Internationalizing Your Program

This chapter describes how to:

- Create a source physical file with a specific Coded Character Set Identifier (CCSID)
- Change the CCSID of a member in a source physical file to the CCSID of another member in another source physical file
- Convert the CCSID for specific source statements in a member.

The ILE C/C++ compiler recognizes source code that is written in most single-byte EBCDIC CCSID (Coded Character Set Identifier). CCSID 290 is not recognized because it does not have the same code points for the lowercase letters a to z. All of the other EBCDIC CCSIDs do have the same code points for the lowercase letters a to z. String literals can be converted back to CCSID 290 by using the `#pragma convert` directive. A file with CCSID 290 still compiles because the ILE C/C++ compiler converts the file to CCSID 037 before compiling.

CCSID 905 and 1026 are not recognized because the " character varies on these CCSIDs.

The CRTCMOD/CRTCPPMOD and CRTBNDC/CRTBNDCPP commands do not support the SRCSTMF parameter in a mixed-byte environment.

Double-byte character set (DBCS) source code requires special programming considerations.

Note: You should tag the source physical file with a CCSID value number if the CCSID (determined by the primary language) is other than CCSID 037 (US English).

Coded Character Set Identifier

A **Coded Character Set Identifier (CCSID)** comprises a specific set of an encoding scheme (EBCDIC, ASCII, or 8-bit ASCII), character set identifiers, code page identifiers, and additional coding-related information that uniquely identifies the coded graphic character representation used.

A **character set** is a collection of graphic characters.

Graphic characters are symbols, such as letters, numbers, and punctuation marks.

A **code page** is a set of binary identifiers for a group of graphic characters.

Code points are binary values that are assigned to each graphic character, to be used for entering, storing, changing, viewing, or printing information.

Character Data Representation Architecture (CDRA) defines the CCSID values to identify the code points used to represent characters, and to convert the character data as needed to preserve their meanings.

Source File Conversion

Your ILE C/C++ source program can be made up of more than one source file. You can have a root source member and multiple secondary source files (such as include files and DDS files).

If any secondary source files are tagged with CCSIDs different from the root source member, their contents are automatically converted to the CCSID of the root source member as they are read by the ILE C/C++ compiler.

If the primary source physical file has CCSID 65535, the job CCSID is assumed for the source physical file. If the source physical file has CCSID 65535 and the job is CCSID 65535, and the system has non-65535, the system CCSID value is assumed for the source physical file. If the primary source physical file, job, and system have CCSID 65535, then CCSID 037 is assumed. If the secondary file, job, and system CCSID is 65535, then the CCSID of the primary source physical file is assumed, and no conversion takes place.

The compiler converts DBCS source files to CCSID 037.

Creating a Source Physical File with a Coded Character Set Identifier

You specify the character set you want to use with the CCSID parameter when you create a source physical file. The default for the CCSID parameter is the CCSID of the job.

This figure shows you what happens when you create a program object that has a root source member with CCSID 273 and include files with different CCSIDs. The ILE C compiler converts the include files to CCSID 273. The program object is created with the same CCSID as the root source member.

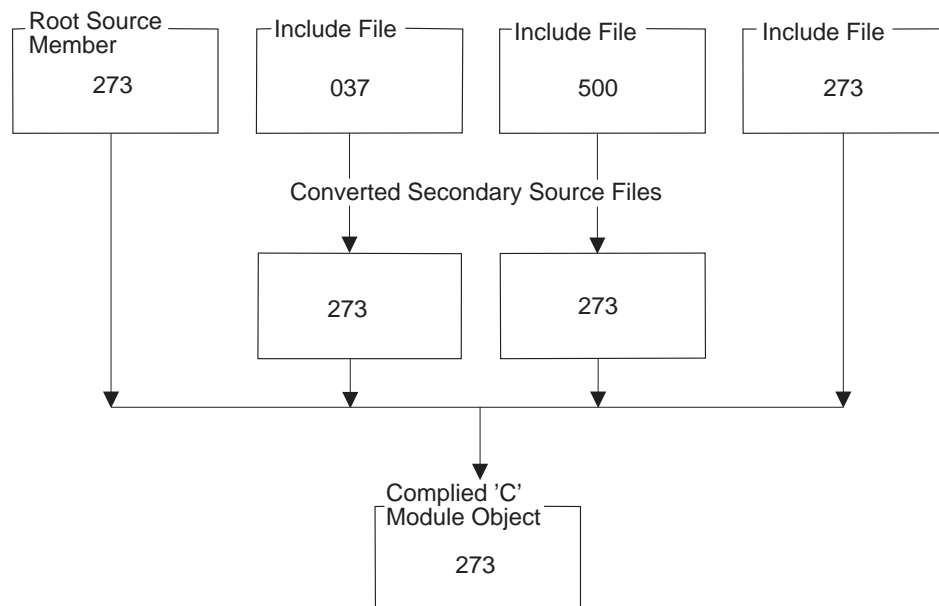


Figure 202. Source File CCSID Conversion

Note: Some combinations of the root source member CCSID and the include file CCSID are not supported.

Example

The following example shows you how to specify CCSID 273 for the source physical file QCSRC in library MYLIB.

To create a source physical file with CCSID 273, type:

```
CRTSRCPF FILE(MYLIB/QCSRC) CCSID(273)
```

Changing the Coded Character Set Identifier (CCSID)

You can change the CCSID of a member within one source physical file to the CCSID of a member in another source physical file by using the Copy File (CPYF) command with option FMTOPT(*MAP) and MBROPT(*ADD). During the copy file operation the FROMFILE member is converted to the CCSID of the TOFILE member.

Example

The following example shows you how to change the first member in a source file with CCSID 037 to CCSID 273. The source file NEWCCSID with CCSID 037 must exist with a member.

To change the CCSID, type:

```
CPYF FROMFILE(MYLIB/NEWCCSID) TOFILE(MYLIB/QCSRC) FMTOPT(*MAP) MBROPT(*ADD)
```

The first member in the file NEWCCSID is copied to QCSRC with CCSID 273.

Note: If CCSID 65535 or *HEX is used, it indicates that character data in the fields is treated as bit data and is not converted.

Converting String Literals in a Source File

You can convert the string literals in a source program from the point that the #pragma convert directive is specified to the end of the program. The #pragma convert directive specifies the CCSID to use for converting the string literals from that point onward in the program. The conversion continues until the end of the source or until another #pragma convert directive is specified.

If a CCSID with the value 65535 is specified, the CCSID of the root source member is assumed. If the source file CCSID value is 65535, CCSID 037 is assumed. The CCSID of the string literals before conversion is the same CCSID as the root source member. The CCSID can be either EBCDIC or ASCII.

Example

The following example shows you how to convert the string literals in T1520CCS to ASCII CCSID 850 even though the CCSID of the source physical file is EBCDIC.

Note: In this example, the TGTCCSID parameter is defaulted to *SOURCE.

1. Type:

```
CRTBNDC PGM(MYLIB/T1520CCS) SRCFILE(QCLE/QACSRC)
```

To create the program T1520CCS using the following source:

```

/* This program uses the #pragma convert directive to convert      */
/* string literals.                                              */
#include <stdio.h>
char EBCDIC_str[20] = "Hello World";
#pragma convert(850)          /* Use the #pragma convert      */
                              /* directive to convert the */
                              /* string to ASCII, CCSID 850. */

char ASCII_str[20] = "Hello World";
#pragma convert(0)           /* Stop string conversion. */
int main(void)
{
    int i;
    printf ("EBCDIC_str(hex) = "); /* Print hex value of EBCDIC */
    for (i = 0; i < 11; ++i)      /* string.                  */
        printf ("%X ",EBCDIC_str[i]);
    printf ("\n\n");
    printf ("ASCII_str(hex) = "); /* Print hex value of ASCII */
    for (i = 0; i < 11; ++i)      /* string.                  */
        printf ("%X ",ASCII_str[i]);
}

```

Figure 203. T1520CCS — ILE C Source to Convert Strings and Literals

The CRTBNDC command creates the program T1520CCS in library MYLIB. Program T1520CCS converts the EBCDIC string Hello World to ASCII CCSID 850.

2. To run the program T1520CCS, type:

```
CALL PGM(MYLIB/T1520CCS)
```

The output is as follows:

```

EBCDIC_str(hex) = C8 85 93 93 96 40 E6 96 99 93 84
ASCII_str(hex) = 48 65 6C 6C 6F 20 57 6F 72 6C 64
Press ENTER to end terminal session.

```

Using Unicode Support for Wide-Character Literals

The ILE C/C++ compiler includes support for Unicode character storage. Wide-character literals and strings can be stored as UCS-2 characters (Unicode CCSID 13488), minimizing the need for code page conversions when developing applications for international use. Using Unicode permits processing of characters in multiple character sets without loss of data integrity.

To enable Unicode character set support, specify LOCALETYP(*LOCALEUCS2) on the CRTCMOD/CRTCMODCPP or CRTBNDC/CRTBNDCPP command line. When this option is selected, the `__UCS2__` macro is defined. Wide-character literals are interpreted using the CCSID of the root source file, then translated to the Unicode CCSID (13488) when stored.

Wide-character literals can be represented in various ways. These representations and how they are handled are described below:

Trigraphs

Trigraph characters used as literals are converted to their corresponding Unicode characters. For example:

```
wchar_t *wcs = L" ??(";
```

is equivalent to:

```
wchar_t wcs[] = {0x0020, 0x005B, 0x0000};
```

Character Escape Codes (\a, \b, \f, \n, \r, \t, \v, \', \", \?)

Character escape codes are converted to their corresponding Unicode escape codes. For example:

```
wchar_t *wcs = L" \t \n";
```

is equivalent to:

```
wchar_t wcs[] = {0x0020, 0x0009, 0x0020, 0x000A, 0x0000};
```

Numeric Escape Codes (\xnnnn, \ooo)

Numeric escape codes are not converted to Unicode. Instead, the numeric portion of the literal is preserved and assumed to be a Unicode character represented in hexadecimal or octal form. For example:

```
wchar_t *wcs = L" \x4145";
```

is equivalent to:

```
wchar_t wcs[] = {0x0020, 0x4145, 0x0000};
```

Specifying \xnn in a wchar_t string string literal is equivalent to specifying \x00nn.

Hexadecimal constant values larger than 0xFF are normally considered invalid. Setting the *LOCALEUCS2 option changes this to allow 2-byte hexadecimal initialization of wchar_t types only. For example:

```
wchar_t wc = L'\x4145'; /* Valid only with *LOCALEUCS2 option, */
                        /* otherwise an out of bounds error */
                        /* will result. */

char      c = '\x4145'; /* Not valid due to size restriction. */
                        /* Error will result with or without */
                        /* specifying *LOCALEUCS2 option. */
```

Note: Numeric hexadecimal escape codes are not validated other than to ensure type size-limits are not exceeded.

DBCS Characters

DBCS characters entered as hexadecimal escape sequences are not converted to Unicode. They are stored as received.

Effect of Unicode on #pragma convert() Operations

When LOCALETYPE(*LOCALEUCS2) is specified, wide-character literals will always represent a UCS-2 character literal regardless of CCSID used by the root source file. As a result, #pragma convert() will ignore wide-character literals when converting characters from one code page to another.

Example

This example assumes a CCSID 37 source:

```
#include <stdio.h>
#include <wchar.h>
void main () {
#pragma convert (500)
    wchar_t wcs1[] = L"[]";
    char    str1[] = "[]";
#pragma convert (0)
    wchar_t wcs2[] = L"[]";
```

```

char    str2[] = "[]";
printf("str1 = %x %x\n", str1[0], str1[1]);
printf("str2 = %x %x\n", str2[0], str2[1]);
printf("wcs1 = %04x %04x\n", wcs1[0], wcs1[1]);
printf("wcs2 = %04x %04x\n", wcs2[0], wcs2[1]);
}

```

Running the program would result in output similar to that shown below.

```

str1 = 4a 5a
str2 = ba bb
wcs1 = 005b 005d
wcs2 = 005b 005d

```

Target CCSID Support

Target CCSID (TGTCCSID) is a parameter used with the CRTCMOD/CRTCPMOD and CRTBNDC/CRTBNDCPP ILE C/C++ compiler commands. TGTCCSID allows the compiler to process source files from a wide variety of CCSIDs or codepages (in the case of a source stream file), and target a module CCSID different from that of the root source file, as long as the translation between the source character set and the target module CCSID is installed into the operating system.

The source CCSID is provided to the compiler via the source file's character set. The target CCSID is not dependent on the source's CCSID, but variable, requiring the TGTCCSID parameter in the compiler's commands. TGTCCSID allows you to enter the desired target module CCSID. When the TGTCCSID differs from the source file's CCSID, the source files are converted to the TGTCCSID and processed. This ensures that the target module and all its character data components (for example, listing, string pool) are in the desired TGTCCSID. You can then develop in one character set and target another. The argument defaults to the source file's character set so the default behavior is backward compatible (with the exception of 290, 930 and 5026).

Providing support for more source character sets, increases the NLS usability of the compilers. CCSIDs 290, 930 and 5026 are now supported. The TGTCCSID parameter provides solutions to more complex NLS programming issues. For example, several modules with different module CCSIDs may be compiled from the same source by simply recompiling the source with different TGTCCSID values.

Literals, Comments, and Identifiers

The TGTCCSID parameter allows you to choose the CCSID of the resulting module. The module's CCSID identifies the coded character set identifier in which the module's character data is stored, including character data used to describe literals, comments and identifier names described by the source (with the exception of identifier names for CCSIDs 5026, 930 and 290).

For example, if the root source file has a CCSID of 500 and the compiler parameter TGTCCSID default value is not changed (i.e., *SOURCE), the behavior is as before with the resulting module CCSID of 500. All string and character literals, both single and wide-character, are as described in the source file's CCSID. Translations may still occur as before for literals, comments and identifiers.

However, if the TGTCCSID parameter is set to 37 and the same source recompiled, the resulting module CCSID is 37; all literals, comments, and identifiers are translated to 37 where needed and stored as such in the module.

Regardless of what CCSID the root source and included headers are in, the resulting module is defined by the TGTCCSID, and all of its literals, comments, and identifiers are stored in this CCSID.

Restrictions

Debug Listing View

The introduction of the TGTCCSID parameter removes the restriction preventing the compilation of source with CCSIDs 5026, 930 or 290 without the loss of DBSC characters in literals and comments. However, a lesser restriction is introduced for these CCSIDs; when using listing view to debug a module compiled with TGTCCSID equal to CCSID 5026, 930, or 290, substitution characters appear for all characters not compatible with CCSID 37.

Format Strings

When coding format strings for C runtime I/O functions (for example, `printf("%d\n", 1234);`) the format string must be compatible with CCSID 37. When targetting CCSIDs 290, 930, 5026 which are not CCSID 37 compatible, a `#pragma convert(37)` is required around the format string literal to ensure the runtime function processes the format string correctly.

Valid Target Encoding Schemes

TGTCCSID values are restricted to CCSIDs with encoding schemes 1100 or 1301. An error message is issued by the command if any other value is entered.

1100 = EBCDIC, single-byte, No code extension is allowed, Number of States = 1.

1301 = EBCDIC, mixed single-byte and double-byte, using shift-in (SI) and shift-out (SO) code extension method, Number of States = 2.

Chapter 19. International Locale Support

International locale support allows programs to change their behavior according to the user's language environment. This support has three key components:

- programming tools that create language-specific data
- programming interfaces (functions) that allow access to this data
- methods of creating programs that are automatically sensitive to the language environment in which they run.

Elements of a Language Environment

The typical elements of a language environment are as follows:

- Native language:
The natural language of the user.
- Character sets and coded character sets:
A coded character set is created by mapping the characters of a character set onto a set of code points (hexadecimal values). See Chapter 18, "Internationalizing Your Program" on page 403 for more information about coded character sets and CCSIDs.
- Collating and ordering:
The relative order of characters that are used for sorting.
- Character classification:
The type of a character (for example, alphabetic, numeric) in a character set.
- Character case conversion:
The mapping between uppercase and lowercase characters in a character set.
- Date and time format:
The format of date and time data (for example, order of the months, names of the weekdays).
- Format of numbers and monetary quantities:
The format of numbers and monetary quantities. For example, numeric grouping, decimal-point character, and monetary symbols.
- Format of affirmative and negative system responses:
The format of affirmative and negative system responses.

Locales

A locale is a system object that specifies how language-specific data is processed, printed and displayed. A locale is made up of categories that describe the character set, collating sequence, date and time representation and monetary representation of the language environment in which it will be used. Using locales and locale-sensitive functions, applications can be created that are independent of language, cultural data, or character set, yet are sensitive to the language environment of the user.

ILE C/C++ Support for Locales

The ILE C/C++ compiler and run time supports locales of type *CLD and *LOCALE. Locales of type *LOCALE and the ILE C/C++ support for them is based on the IEEE POSIX P1003.2, ISO/IEC 9899:1990/Amendment 1:1994[E], and X/Open Portability Guide standards for global locales and coded character set conversion.

The POSIX standard defines a much more comprehensive set of functions and locale data for application internationalization that is compared to that available for *CLD locales. By supporting the POSIX specification for locales in the ILE C/C++ run time and introducing new functions which comply with the XPG4, POSIX and ISO/IEC standards, ILE C/C++ programs using locales of type *LOCALE become more portable to and from other operating systems.

Note: CLD is for ILE C use only.

ILE C/C++ Support for *CLD and *LOCALE Object Types

Programs that were compiled prior to V3R7 use the *CLD locale support. Programs that are compiled with the option LOCALETYPE(*CLD) on the CRTCMOD or CRTBNDC command uses the locale support that is provided by ILE C/C++ for *CLD objects. Programs compiled with the option LOCALETYPE(*LOCALE) on the CRTCMOD/CRTCMODCPP or CRTBNDC/CRTBNDCPP command uses the locale support provided by ILE C/C++ for locales of type *LOCALE.

If you wish to convert your application from using locales of type *CLD to locales of type *LOCALE, the only changes required to your C source code are in calls to `setlocale()`. However, there are many differences between the locale definition source for *CLD and *LOCALE objects. The *LOCALE definition source members for many language environments are provided by the system in the optionally installable library QSYSLOCALE. You may also convert your existing *CLD locale source to the *LOCALE source definition. See Table 29 for a mapping of the commands in a source file for creating *CLD objects to the corresponding keywords in a source file for creating *LOCALE objects.

An application may use either locales of type *CLD or *LOCALE, but not both. If an ILE C program attempts to use both types of locales, the results are undefined. ILE C++ does not use *CLD. Also, some locale-sensitive functions are only supported when locales of type *LOCALE are used. See Table 31 on page 419 for a list of locale-sensitive functions.

C Locale Migration Table

The ILE C run time supports two implementations of the `setlocale()` function and the locale-sensitive functions. The original implementation uses locale objects of type *CLD, while the second implementation uses locale objects of type *LOCALE. The following table summarizes the differences in the locale source keywords between the locales of type *CLD and *LOCALE.

Table 29. C Locale Migration Table

Category	*CLD Command	Description and Format	*LOCALE Keyword
LC_TIME	AM	String of characters used to represent the locale's equivalent of AM.	am_pm

Table 29. C Locale Migration Table (continued)

Category	*CLD Command	Description and Format	*LOCALE Keyword
LC_CTYPE	CHARTYP	Character set type.	Determined by CCSID of the locale
LC_COLLATE	CPYSYSCOL	System collating sequence table.	cpysyscol
LC_COLLATE	COLLSTR	String transformation table.	collating-element
LC_COLLATE	COLLTAB	Character weight reassignment for the strcoll function.	collating-element
LC_CTYPE	CTYPE	Set the attributes of a particular character.	upper lower alpha digit space cntrl punct graph print xdigit blank
LC_MONETARY	CURR	A string of characters that represent the currency symbol.	currency_symbol
LC_TIME	DATFMT	A string of characters used to specify the format of the date in this locale.	d_fmt
LC_TIME	DATTIM	A string of characters used to specify the format of the date and time in this locale.	d_t_fmt
LC_NUMERIC	DEC	Decimal point character for formatted non-monetary quantities.	decimal_point
LC_TOD	DSTEND	The instant when Daylight Savings Time ceases to be in effect. (day,time)	dstend
LC_TOD	DSTNAME	The name of the time zone when Daylight Savings Time is in effect.	dstname
LC_TOD	DSTSHIFT	The number of seconds that the locale's time is shifted when Daylight Savings Time takes effect.	dstshift
LC_TOD	DSTSTART	The instant when Daylight Savings Time comes into effect.	dststart
LC_NUMERIC	GROUP	Digit grouping from processing digits to the left of the decimal point from left to right for formatted non-monetary quantities.	grouping
LC_MONETARY	ICURR	String of characters used to represent the currency symbol in an internationally formatted monetary quantity.	int_curr_symbol
LC_TIME	LDAYS	The long form of each day of the week.	day
LC_TIME	LMONS	The long form of each month of the year.	mon

Table 29. C Locale Migration Table (continued)

Category	*CLD Command	Description and Format	*LOCALE Keyword
LC_CTYPE	LOWER	Set the lowercase character to be returned for a given character by the tolower library function.	tolower
LC_MONETARY	MDEC	String of characters used for the decimal point in a formatted monetary quantity.	mon_decimal_point
LC_MONETARY	MFDIGIT	Number of fractional digits to display in a formatted monetary quantity.	frac_digits
LC_MONETARY	MGROUP	Digit grouping from processing digits to the left of the decimal point from left to right for formatted monetary quantities.	mon_grouping
LC_MONETARY	MIFDIGIT	Number of fractional digits to display in an internationally formatted monetary quantity.	int_frac_digits
LC_MONETARY	MMINUS	String of characters used to represent a negative value in a formatted negative monetary quantity.	negative_sign
LC_MONETARY	MMINUSPOS	An encoded value used to represent the position of the negative symbol in a formatted negative monetary quantity.	n_sign_posn
LC_MONETARY	MNCSP	A true or false value used to determine if the currency symbol precedes the value in a formatted negative monetary quantity. If the value is false, then the symbol succeeds the value.	n_cs_precedes
LC_MONETARY	MNSBYS	A true or false value used to determine if the currency symbol is space-separated in a formatted negative monetary quantity.	n_sep_by_space
LC_MONETARY	MPCSP	A true or false value used to determine if the currency symbol precedes the value in a formatted positive monetary quantity. If the value is false, then the symbol succeeds the value.	p_cs_precedes
LC_MONETARY	MPLUS	String of characters used to represent a positive value in a formatted positive monetary quantity.	positive_sign

Table 29. C Locale Migration Table (continued)

Category	*CLD Command	Description and Format	*LOCALE Keyword
LC_MONETARY	MPLUSPOS	An encoded value used to represent the position of the positive symbol in a formatted positive monetary quantity.	p_sign_posn
LC_MONETARY	MPSBYS	A true or false value used to determine if the currency symbol is space-separated in a formatted positive monetary quantity.	p_sep_by_space
LC_MONETARY	MSEP	Character used to separate grouped digits in a formatted monetary quantity.	mon_thousands_sep
LC_TIME	PM	String of characters used to represent the locale's equivalent of PM.	am_pm
LC_TIME	SDAYS	The short form of each day of the week.	abday
LC_NUMERIC	SEP	Character used to separate grouped digits in a formatted non-monetary quantity.	decimal_point
LC_TIME	SMONS	The short form of each month of the year.	abmon
LC_TIME	TIMFMT	A string of characters used to specify the format of the time in this locale.	t_fmt
LC_TOD	TNAME	String of characters used to represent the locale's time zone name.	tname
LC_TOD	TZDIFF	The number of minutes that the locale's time zone is different from Greenwich Mean Time.	tzdiff
LC_CTYPE	UPPER	Set the uppercase character to be returned for a given character by the toupper library function.	toupper

POSIX Locale Definition and *LOCALE Support

Locale definition source files that conform to the IEEE POSIX P1003.2 standard will be shipped with the system in the optionally installable library QSYSLOCALE. One *LOCALE object, the C locale as defined by the POSIX standard, is provided with the system. Other locales of type *LOCALE can be created with the CRTLOCALE command from the locale source definition members in the QSYSLOCALE library.

LOCALETYPE Compiler Option

The LOCALETYPE option on the CRTCMOD/CRTCPCPPMOD or CRTBNDC/CRTBNDCPP command allows a program to specify the type of locale object to be used when it is being compiled. The keyword options for the LOCALETYPE option are *CLD and *LOCALE, with the default being *LOCALE. The keyword *CLD enables the *CLD locale support, whereas the keyword *LOCALE enables the support for locales of type *LOCALE.

The CL command format for enabling the run time that support locales of type *LOCALE is:

```
CRTCMOD MODULE(MYLIB/MYMOD) SRCFILE(MYLIB/QCSRC) LOCALETYPE(*LOCALE)
CRTBNDC PGM(MYLIB/MYPPGM) SRCFILE(MYLIB/QCSRC) LOCALETYPE(*LOCALE)
```

When the *LOCALE keyword is specified for the LOCALETYPE option, the ILE C/C++ compiler defines the macro `__POSIX_LOCALE__`. When `__POSIX_LOCALE__` is defined, the locale-sensitive C run-time functions are remapped to functions that are sensitive to locales that are defined in *LOCALE objects. In addition, certain ILE C/C++ run-time functions can only be used with locales of type *LOCALE and do not work with *CLD locales. These functions are available only in V3R7 and later releases of the ILE C/C++ runtime. The list of locale-sensitive functions later in this chapter indicates which functions are sensitive only to locales of type *LOCALE.

Note: The default has changed. Prior to V5R1, *CLD was the default value for ILE C. As of V5R1, the default has been changed to *LOCALE.

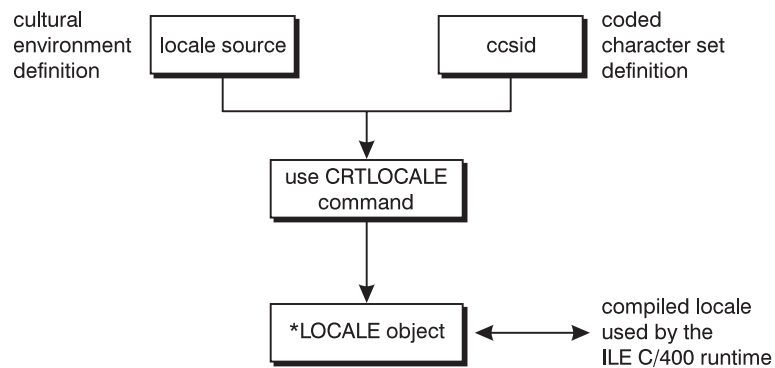
Creating Locales

On iSeries systems, *LOCALE objects are created with the CRTLOCALE command, specifying the name of the file member containing the locale's definition source, and the CCSID to be used for mapping the characters of the locale's character set to their hexadecimal values.

A locale definition source member contains information about a language environment. This information is divided into a number of distinct categories which are described in the next section. One locale definition source member characterizes one language environment.

Characters are represented in a locale definition source member with their symbolic names. The mapping between the symbolic names, the characters they represent, and their associated hexadecimal values is based on the CCSID value that is specified on the CRTLOCALE command.

Here is a model of how a locale of type *LOCALE is created:



Note: There is no support for locales created with non-EBCDIC CCSIDs.

Creating modules using **LOCALETYPE(*LOCALE)**

When you create modules with the **LOCALETYPE(*LOCALE)** option, **MB_CUR_MAX** have the following values:

- 1 for locales built with a single byte CCSID, such as 00037
- 4 for locales built with a mixed byte CCSID such as 00939.

MB_CUR_MAX is dependent on the **LC_CTYPE** category of the current locale.

Categories Used in a Locale

A locale and its definition source member contain the following categories.

Table 30. Categories Used in a Locale

Category	Purpose
LC_COLLATE	Defines the collation relations among the characters. Affects the behavior of the collating functions <code>strcoll()</code> , <code>strxfrm()</code> , <code>wscoll()</code> and <code>wcsxfrm()</code> .
LC_CTYPE	Defines character types, such as upper-case, lower-case, space, digit, and punct. Affects the behavior of character handling functions.
LC_MESSAGES	Defines the format and values for responses from the application.
LC_MONETARY	Defines the monetary names, symbols, punctuation, and other details. Affects monetary information returned by <code>localeconv()</code> .
LC_NUMERIC	Defines the decimal-point (radix) character for the formatted input/output and string conversion functions, and the non-monetary formatting information returned by <code>localeconv()</code> .
LC_TIME	Defines the date and time conventions, such as calendar used, time zone, and days of the week. Affects the behavior of time display functions.
LC_TOD	Defines time zone difference, time zone name, and Daylight Savings Time start and end.

Setting an Active Locale for your Application

All C and C++ applications using locales of type ***LOCALE** have an active locale which is scoped to the activation group of the program. The active locale determines the behavior of the locale-sensitive functions in the C library. The active

locale can be set explicitly with a call to `setlocale()`. See the *ILE C for AS/400 Run-Time Library Reference* for more information on using `setlocale()`.

If the active locale is not set explicitly by a call to `setlocale()`, it is implicitly set by the C run time at program activation time. Here is how the C run time sets the active locale when a program is activated:

- If the user profile has a value for the LOCALE parameter other than *NONE (the default) or *SYSVAL, that value is used for the application's active locale.
- If the value of the LOCALE parameter in the user profile is *NONE, the default "C" locale becomes the active locale.
- If the value of the LOCALE parameter in the user profile is *SYSVAL, the locale associated with the system value QLOCALE will be used for the program's active locale.
- If the value of QLOCALE is *NONE, the default "C" locale becomes the active locale.

Using Environment Variables to Set the Active Locale

A program's active locale is set either implicitly at program startup, as described above, or explicitly by a call to `setlocale()`. The `setlocale()` function takes two arguments: an integer representing the locale category whose values are needed for the active locale, and the name of the locale from which the values are to be taken. The name of the locale can be either "C", "POSIX", the fully-qualified path name of a locale object of type *LOCALE, or a null string("").

When the locale argument of `setlocale()` is specified as a null string(""), `setlocale()` sets the active locale according to the environment variables defined for the job in which the program is running. You can create environment variables that have the same names as the locale categories and specify the locale to be associated with each environment variable. The LANG environment variable is automatically created during job initiation when you specify a locale path name for the LOCALE parameter in your user profile or for the QLOCALE system value.

When a program calls `setlocale(category, "")`, the locale-related environment variables defined in the current job are checked to find the locale name or names to be used for the specified category. The locale name is chosen according to the first of the following conditions that applies:

1. If the environment variable LC_ALL is defined and is not null, the value of LC_ALL is used for the specified category. If the specified category is LC_ALL, that value is applied to all categories.
2. If the environment variable for the category is defined and is not null, then the value that is specified for the environment variable is used. For the LC_ALL category, if individual environment variables (for example, LC_CTYPE, LC_MONETARY, and so on) are defined and are not null, then their values are used for the categories that correspond to the environment variables. This could result in the locale information for each category that is retrieved from a different locale object.
3. If the environment variable LANG is defined and is not null, the value of the LANG environment variable is used.
4. If no non-null environment variable is present to supply a locale value, the default "C" locale is used.

If the locale specified for the environment variable is found to be invalid or non-existent, `setlocale()` returns NULL and the program's active locale remains unchanged.

For `setlocale(LC_ALL, "")`, if the locale names found identify valid locales on the system, `setlocale()` returns a string naming the locale associated with each locale category. Otherwise, `setlocale()` returns NULL, and the program's locale remains unchanged.

SAA and POSIX *Locale Definitions

If an ILE C program is compiled with `LOCALETYPE(*LOCALE)` and `setlocale()` is not called or if it is called with locale name "C" or "POSIX", the default "C" environment used is that specified in the "POSIX" locale definition source in the QSYSLOCALE library. This locale definition is slightly different from the default "C" locale for type *CLD. Another locale definition source member that is called "SAA" is provided in the QSYSLOCALE library for compatibility with the default "C" locale of type *CLD.

If you wish to migrate your application from locales of type *CLD to locales of type *LOCALE, but you want to be compatible with the default "C" locale of type *CLD, use the "SAA" locale definition source member in the QSYSLOCALE library to create a locale with the CRTLOCALE command. Then use the name of this locale when you call `setlocale()` in your application.

The differences between the "SAA" and "POSIX" locale definitions are as follows:

- For the LC_CTYPE category, the "SAA" locale has all the EBCDIC control characters defined in the 'cntrl' class, whereas the "POSIX" locale only includes the ASCII control characters. Also, "SAA" has the cent character and the broken vertical line as 'punct' characters whereas "POSIX" does not include these two characters in its 'punct' characters.
- For the LC_COLLATE category, the default collation sequence for "SAA" is the EBCDIC sequence whereas "POSIX" uses the ASCII sequence. This is independent of the CCSID mapping of the character set. For the "POSIX" locale, the first 128 ASCII characters are defined in the collation sequence, and the remaining EBCDIC characters are at the end of the collating sequence. Also, in the "SAA" locale definition, the lowercase letters collate before the uppercase letters, whereas in the "POSIX" locale definition, the lowercase letters collate after the uppercase letters.
- For the LC_TIME category, the "SAA" locale specifies the date and time format (`d_t_fmt`) as "%Y/%M/%D %X" whereas the "POSIX" locale uses "%a %b %d %H %M %S %Y".

Locale-Sensitive Run-Time Functions

The following ILE C/C++ run-time functions are sensitive to locales:

Table 31. Locale-Sensitive Run-Time Functions

<code>asctime()</code>	<code>asctime_r()</code>	<code>btowc()</code> ¹
<code>ctime()</code>	<code>ctime_r()</code>	<code>fprintf()</code>
<code>fgetwc()</code> ¹	<code>fgetws()</code> ¹	<code>fputwc()</code> ¹
<code>fputws()</code> ¹	<code>fwprintf()</code> ¹	<code>fwscanf()</code> ¹
<code>getwc()</code> ¹	<code>getwchar()</code> ¹	<code>gmtime()</code>

Table 31. Locale-Sensitive Run-Time Functions (continued)

gmtime_r()	isalnum() to isxdigit()	isascii()
iswalnum() to iswxdigit() ¹	localeconv()	localtime()
localtime_r()	mblen()	mbrlen() ¹
mbsinit() ¹	mbrtows() ¹	mbsrtowcs() ¹
mbstowcs() ¹	mbtowc() ¹	mktime()
nl_langinfo() ¹	printf()	putwc() ¹
putwchar() ¹	regcomp()	regerror()
regexec()	regfree()	setlocale()
sprintf()	strcoll()	strfmon() ¹
strftime()	strptime() ¹	strxfrm()
swprintf() ¹	swscanf() ¹	time()
toascii()	tolower()	toupper()
tolower() ¹	towtrans() ¹	towupper() ¹
vfprintf()	vfwprint() ¹	vprintf()
vsprintf()	vswprintf() ¹	vwprintf() ¹
wcrtomb() ¹	wscoll() ¹	wcsftime()
wcstod()	wcstol()	wcstoul()
wcsrtombs() ¹	wcstombs()	wcswidth() ¹
wcsxfrm() ¹	wctob()	wctomb() ¹
wctrans() ¹	wctype() ¹	wcwidth() ¹
wscanf() ¹	wprintf()	
Note: ¹ sensitive to *LOCALE objects only.		

Part 8. Appendixes

Appendix A. Improving Program Performance

Often the cause of a program's poor performance is very specific. Inefficient code can affect performance when it is looped through many times during program execution. When you examine a program to improve performance, look at those aspects which have a significant impact every time a program is run.

Often run-time performance can be improved through minor changes to your source programs. The amount of improvement each tip provides depends on how your program is organized, and on the functions and language constructs your program uses. Some tips may provide substantial performance improvement to your program, while others may offer almost no improvement. Some tips may contradict each other because they may trade off one resource for another. One tip is to reduce the size of the call stack by using static and global variables. Another tip is to improve execution startup performance by reducing the use of static and global variables.

Use performance analysis tools to find out where your performance problems are, and then try and apply different appropriate tips to try and achieve the best performance for your program.

This section describes:

- Data Types
- Classes
- Performance Management
- Exception Handling
- Function Call Performance
- Input and Output Considerations
- Open Pointers
- Shallow Copy and Deep Copy
- Space Considerations
- Compile-Time Performance Tips
- Run-Time Limits

Note: Before trying different tips, first compile and benchmark your programs using full optimization.

Data Types

There are several ways to improve performance through data types. Replacing bit fields with other data types and minimizing the use of static and global variables are some of the ways.

Avoid Using the Volatile Qualifier

Only use the `volatile` qualifier when necessary. `Volatile` specifies that a variable can be changed at any time, possibly by an external program, and therefore it is not a candidate for optimization.

Replace Bit Fields with Other Data Types

Avoid using bit-fields, since it takes more time to access bit-fields than other data types such as short and int. Whenever possible, replace bit-fields with other data types. If a bit-field takes 16 bits and aligns on 2-byte boundary, you can replace it with the short data type.

Note: You can still obtain a run-time improvement if the bit-field is smaller than the integral type. The extra time required for bit-field manipulation code offsets the performance gain due to space saved in data.

Minimize the Use of Static and Global Variables

Minimize the use of static and global variables, if possible. These are initialized whether or not you explicitly initialize them. By not using static and global variables, the performance improvement is obtained at activation group startup.

Use the Register Storage Class

Use the register storage class for a variable that is frequently used. Do not overuse the register storage class, so that the optimizer can place the most frequently used variables into the available hardware registers.

If you use the register storage class, you cannot rely on the value displayed from within the debugger, since you may be referencing an older value that is still in storage.

Classes

When you use the IBM Open Class Library or IBM Access Class Library for OS/400 to create classes, use a high level of abstraction. After you establish the type of access to your class, you can create more specific implementations. This can result in improved performance with minimal code changes. See the *VisualAge for C++ for AS/400 IBM Open Class Library User's Guide* for information on using classes.

When you define structures or data members within a class, define the largest data types first to align them on the largest natural boundary. Define pointers first to reduce the padding necessary to align them on quadword (16-byte) boundaries. Follow them, in order, with the double-word, and half-word items to avoid padding or improve load/store time.

Performance Measurement

You can use a native compiler option to include performance hooks in your generated code.

Use a Compiler Option to Enable Performance Measurement

The performance-measurement compiler option `ENBPFCOL()` allows you to specify whether or not the compiler should generate code (sometimes called "performance hooks") into your compiled program or module. The performance hooks enable the Performance Explorer to analyze your programs. The default for this option specifies that program entry procedure level performance-measurement code is generated for your compiled module or program.

Compiling performance collection code into the module or program allows performance data to be gathered and analyzed. The insertion of the additional collection code results in slightly larger module/program objects and may affect performance to a small degree.

Types of performance data collected include:

- Pre- and post-call information
This information is gathered immediately before and after calling any given functions. It provides a record of where a call was made, and information on the performance of the operation called.
- Procedure entry and exit information
This information is gathered immediately upon entry into a procedure and exit from that procedure. A snapshot is taken of the current performance statistics when entering a procedure, and a calculation is made of the differences in those statistics when exiting that procedure.

When performance collection code is generated into a leaf procedure, the procedure is changed so that it is no longer a leaf procedure. (A leaf procedure is one that does not call any other procedures.)

The extra expense of capturing data on a leaf procedure is mainly due to the fact that the leaf procedures being hooked lose their 'leaf-ness' in the process. This happens due to the fact that hooks are basically calls to collection routines and leaf procedures by definition do not make calls.

See the *VisualAge for C++ for AS/400 C++ User's Guide* for information on these options.

Exception Handling

Reducing exceptions, turning off C2M messages during record I/O and using direct monitor handlers are some of ways to improve exception handling performance.

Reduce Exceptions

Exceptions are expensive to process. Try to minimize the number of exceptions in your programs.

If you use record I/O, the `rtncode=y` option can be used on `_Ropen()` to help reduce exceptions. Files that are opened with this option do not have exceptions generated for the "Record not found" (CPF5006) and the "End-of-File" (CPF5001) conditions. When these conditions occur, the `num_bytes` field of the `_RIOFB_T` structure is updated and `errno` is set, but no exceptions are generated. For the "Record not found" condition, this field is set to zero. For the "End-of-File" condition, it is set to EOF.

If your program generates either of these conditions many times, then a significant performance improvement may be seen using this option.

Turn Off C2M Messages during Record Input and Output

Turn off C2M messages during record I/O. This is controlled by the variable `_C2M_MSG` declared in `<recio.h>`. Set the variable `_C2M_MSG` to zero to turn off C2M messages during record I/O. If `_C2M_MSG` is set, record I/O sends C2M messages to your program when it detects these errors: C2M3003, C2M3004, C2M3005,

C2M3009, C2M3014, C2M3015, C2M3040, C2M3041, C2M3042 and C2M3044. Turning it off prevents record I/O sending such messages. Removing data truncation messages with signal handlers or message handlers is no longer necessary when the C2M messages are turned off during record I/O.

Use a Direct Monitor Handler

Use a direct monitor handler (this is done using the `#pragma exception_handler` in C++) instead of a signal handler. When an exception occurs, the compiler first attempts to use any direct monitor handler if there is one. Otherwise, C++ maps the exception to a signal, and calls the corresponding signal handler. By using a direct monitor handler, the signal mapping and search for signal handler are saved. The direct monitor handler should mark the exception as handled, if the exception is one you are expecting; otherwise the exception is percolated again. See the *VisualAge for C++ for AS/400 C++ Language Reference* for information on this pragma.

Avoid Percolating Exceptions

Try to handle an exception in the place it occurs. There is some processing overhead incurred with exception percolation. See Chapter 13, “Handling Exceptions in Your Program” on page 257 for information on handling exceptions.

Function Call Performance

You can reduce the number of function calls by using inline functions or macro expressions. In C++, macro expansion is not recommended. Use the `inline` keyword, and turn on inlining instead.

Reduce the Number of Function Calls and Function Arguments

You can improve performance by changing function calls to inline functions or macro expressions, provided such a change does not increase the size of the program object too much. A program object size increase may cause more page faults which slows the program down. Try to strike a balance between program size and inlining or macro expressions for an optimum level of performance.

When a function is only called in a few places but executed many times, changing the function to an inline function saves many function calls and results in performance improvement.

The `INLINE` compile-time option allows you to request that the compiler replace a function call with that function’s code in place of the function call. If the compiler allows the inlining to take place, the function call is replaced by the machine code that represents the source code in the function definition.

Inlining is a method that allows you to improve the run time performance of a C++ program by eliminating the function call overhead. Inlining allows for an expanded view of the program for optimization. Exposing constants and flow constructs on a global scale allows the optimizer to make better choices during optimization.

The overhead of a bound function call is small compared to the overhead of an external dynamic call. This becomes significant when functions are called many times. Try to avoid external dynamic calls, if possible.

When calling a program dynamically from a C++ program using extern "C" linkage, prototype the program to return void rather than int. Extra processing is involved in accessing the return value of a program call. Passing the address of storage that can hold a return value in the call's argument list is better from a performance viewpoint.

Function call performance can be improved if the system has all of the arguments passed in registers. Since there are only a limited number of registers, in order to increase the chance of having all arguments passed in registers, combine several arguments into a class and pass the address of the class to the function. Since an address is being passed, pass-by-reference semantics are used, which may not have been the case when the arguments were being passed as individual variables.

You can use static class members, a better programming practice that gives better performance.

However, an alternative to passing an argument to a function is to have the variable defined as being global and to have the function use the global variable. Also, global variables can inhibit optimization. Using more global variables increases the amount of work that has to be done at activation group start-up to allocate and initialize the global variables.

Input and Output Considerations

This section covers some Input and Output issues.

Use Record Input and Output Functions

Using record I/O functions instead of stream I/O functions can greatly improve I/O performance. Instead of accessing one byte at a time, record I/O functions access one record at a time.

The two types of record I/O supported by the ILE C/C++ run time are ANSI C record I/O and ILE C record I/O.

ANSI C Record I/O

The following shows that if you use an ANSI C record I/O in your program, you must specify type = record in the open mode parameter of fopen() when you open a file, and you must use the FILE data type.

```

#include <stdio.h>
#define MAX_LEN 80
int main(void)
{
    FILE *fp;
    int len;
    char buf[MAX_LEN + 1];
    fp = fopen("MY_LIB/MY_FILE", "rb, type = record");
    while ((len = fread(buf, 1, MAX_LEN, fp)) != 0)
    {
        buf[len] = "\0";
        printf("%s\n", buf);
    }
    fclose(fp);
}

```

Figure 204. Using ANSI C Record I/O

ILE C Record I/O

If you use ILE C record I/O in your program, you must use the ILE C record I/O functions, for example, functions that begin with `_R`, and you must use the `_RFILE` data type. The example above can be rewritten as follows:

```

#include <stdio.h>
#include <recio.h>
#define MAX_LEN 80
int main(void)
{
    _RFILE *fp;
    _RIOFB_T *iofb;
    char buf[MAX_LEN + 1];
    fp = _Ropen("MY_LIB/MY_FILE", "rr");
    iofb = _Rreadn(fp, buf, MAX_LEN, __DFT);
    while ( iofb->num_bytes != EOF )
    {
        buf[iofb->num_bytes] = "\0";
        printf("%s\n", buf);
        iofb = _Rreadn(fp, buf, MAX_LEN, __DFT);
    }
    _Rclose(fp);
}

```

Figure 205. Using ILE C Record I/O

Use Input and Output Feedback Information

`_RIOFB_T` is a structure that contains I/O feedback information from ILE C record functions, for example, the number of bytes that are read or are written. By default, the ILE C record I/O functions update the fields in `_RIOFB_T` after a record I/O operation is performed.

If your program does not use all these values, you can improve your application's performance by opening a file as shown:


```
fp = _Ropen("MY_LIB/MY_FILE", "rr, riofb = N");
```

Figure 206. I/O Feedback Information

By specifying `riofb = N`, only the `num_bytes` field, the number of bytes read or written in the `_RIOFB_T` structure is updated. If you specify `riofb = Y`, all the fields in the `_RIOFB_T` structure are updated.

Block Records

You can improve record I/O performance by blocking records. When blocking is specified, the first read causes a whole block of records to be placed into a buffer. Subsequent read operations return a record from the buffer until the buffer is empty. At that time, the next block is fetched.

If you wish to block records when the `FILE` data type is used, open the file with `blksize=value` specified, where `value` indicates the block size. If `blksize` is specified with a value of 0, a block size is calculated for you when you open a file.

If you wish to block records when the `_RFILE` data type is used, specify `blkrcd = Y` when you open the file.

Similar rules apply when blocking records for write operations.

Manipulate the System Buffer

You can improve I/O performance of your ILE C programs by performing read and write operations directly to and from the system buffer, without the need for an application-defined buffer. This system access is referred to as locate mode. The following illustrates how to directly manipulate the system buffer when reading a source physical file.

```
fp = _Ropen("MY_LIB/MY_FILE", "rr, blkrcd = Y, riofb = N");

while ( (_Rreadn(fp, NULL, 92, __DFT))->num_bytes != EOF )
{
    printf("%75.75s\n", ((char *) (*(fp->in_buf))) + 12);
}

_Rclose(fp);
```

Figure 207. Using the System Buffer

The example code above prints up to 75 characters of each record that is contained in the file. The second parameter for the `_Rreadn()`, `NULL`, allows you to manipulate the record in the system buffer. An `_RFILE` structure contains the `in_buf` and `out_buf` fields, which point to the system input buffer and system output buffer, respectively. The example above prints each record by accessing the system's input buffer.

Directly manipulating the system buffer provides a performance improvement when you process very long records. It also provides a significant performance improvement when you use Intersystem Communications Function (ICF) files.

Usually, you only need to access the last several bytes in an ICF file and not all the other data in the record. By using the system buffer directly, the data that you do not use for ICF files need not be copied.

The system buffer should always be accessed through the `in_buf` and `out_buf` pointer in the `_RFILE` structure that is located in the `<recio.h>` header file. Unpredictable results can occur if the system buffer is not accessed through the `in_buf` and `out_buf` pointers.

Open Files Once for Both Input and Output

If your application writes data into a file and then reads the data back, you can improve performance by opening the file only once, instead of the usual two times to complete both input and output. The following illustrates how a file is opened twice and closed twice:

```
fp = _Ropen("MY_LIB/MY_FILE", "wr"); /* Output only.*/
/* Code to write data to MY_FILE */
_Rclose(fp);
/* Other code in your application. */
fp = _Ropen("MY_LIB/MY_FILE", "rr"); /* Input only.*/
/* Code to read data from MY_FILE. */
_Rclose(fp);
```

Figure 208. Opening a File Twice

By changing this example to the following, one call to `_Ropen`, and one call to `_Rclose` is saved:

```
fp = _Ropen("MY_LIB/MY_FILE", "ar+"); /* Input and
output.*/
/* Code to write data to MY_FILE. */
/* Other code in your application. */
/* Code to read data from MY_FILE. */
/* Use either _Rreadf or _Rllocate with the option __FIRST. */
_Rclose(fp);
```

Figure 209. Opening a File Once

Reduce the Use of Shared Files

You can improve performance by not opening the same file more than once in an application. You can allocate the file pointers as global (external) variables, opening the files once, and not closing the file until the end of the application.

Reduce the Number of File Opens and Closes

Open and close are very expensive operations. You can improve performance by opening and closing files only as often as necessary. You can use a class to encapsulate I/O operations such as opening the files once, and not closing the file until the end of the program.

Process Tape Files

You can improve the performance of programs that use tape files by using fixed-length record tape files instead of variable-length tape files.

Use Stream Input and Output Functions

Although using ILE C record I/O functions improves performance more effectively than stream I/O functions, there are still ways to improve performance when using stream I/O.

You can use IFS stream files, with performance similar to record I/O functions, by specifying `SYSIFCOPT(*IFSIO)` on the `CRTCMOD` or `CRTBNDC` commands.

You should use the macro version of `getc` instead of `fgetc()` to read characters from a file. See “Reduce the Number of Function Calls and Function Arguments” on page 426. The macro version of `getc()` reads all the characters in the buffer until the buffer is empty. At this point, `getc()` calls `fgetc()` to get the next record.

For the same reason, you should use `putc()` instead of `fputc()`. The macro version of `putc()` writes all the characters in the buffer until the buffer is full. At this point, `putc()` calls `fputc()` to write the record into the file.

Since stream I/O functions cause many function calls; reducing their use in your application improves performance. The following illustrates calls to `printf()`:

```
printf("Enter next item.\n");
      printf("When done, enter 'done'.\n");
```

Figure 210. Using printf()

The two calls to `printf()` can be combined into a single call so that one call is saved as follows:

```
printf("Enter next item.\n"
      "When done, enter 'done'.\n");
```

Figure 211. Using printf() to Reduce Function Calls

Use C++ Input and Output Stream Classes

Use overloaded shift `<<` `>>` operators on the standard streams, instead of their C equivalents.

Use Physical Files Instead of Source Physical Files

To improve performance, you should use physical files instead of source physical files for your data. When a source physical file is used for stream I/O, the first 12 bytes of each record are not visible to your application. They are used to store the record number and update time. These 12 bytes are an extra load that the ILE C stream I/O functions must manipulate. For example, when performing output, these 12 bytes must be initialized to zero. When performing input, these 12 bytes must be fetched even though they are not passed to your application. Since the ILE C stream I/O functions dynamically create a source physical file when opening a text file that does not exist for output, you should create the file as a physical file before you start your application.

Specify Library Names

You should specify the name of the library in which the file resides. If you do not specify a library name when processing a file, the library list is searched for the file. The search time can be lengthy, depending on the number of libraries and the objects that they contain.

Pointers

Using and comparing pointers can impact performance.

Open Pointers

Avoid using open pointers. Open pointers inhibit optimization. Note that pointers to void (void*) are open pointers in ILE C++.

Pointer Comparisons

Since pointers take up 16 bytes of space, pointer comparisons are less efficient than comparisons using other data types. You may want to replace pointer comparisons with comparisons using other data types, such as int.

This is a program that constructs a linked list, processes all the elements in the list, and finally frees the linked list. Each element in the link list holds one record from a file:

```
#include <string.h>
#include <stdlib.h>
#include <recio.h>

#define MAX_LEN 80

struct link
{
    struct link *next;
    char record[MAX_LEN];
};

int main(void)
{
    struct link *start, *ptr;
    _RFILE *fp;
    int i;

    // Construct the linked list and read in records.

    fp = _Ropen("MY_LIB/MY_FILE", "rr, blkrcd = Y");
    start = (struct link *) malloc(sizeof(struct link));
    start->next = NULL;
    ptr = start;
    for ( i = (_Ropnfbk(fp))->num_records; i > 0; --i )
    {
        _Rreadn (fp, NULL, MAX_LEN, __DFT);
        ptr->next = (struct link *) malloc(sizeof(struct link));
        memcpy(ptr->record, (void const *) *(fp->in_buf), MAX_LEN);
        ptr->next = NULL;
    }
    ptr = start->next;
    free(start);
    start = ptr;
    _Rclose(fp);

    // Process all records.

    for ( ptr = start; ptr != NULL; ptr = ptr->next )
```

```

{
// Code to process the element pointed to by ptr.
}

// Free space allocated for the linked list.

while ( start != NULL )
{
    ptr = start->next;
    free(start);
    start = ptr;
}
}

```

In the preceding program, pointer comparisons are used when processing elements and freeing the linked list. The program can be rewritten using a short type member to indicate the end of the link list. As a result, you change pointer comparisons to integer comparisons:

```

#include <string.h>
#include <stdlib.h>
#include <recio.h>

#define MAX_LEN 80
int i;

struct link
{
    struct link *next;
    short last;
    char record[MAX_LEN];
};

int main(void)
{
    struct link *start, *ptr;
    _RFILE *fp;

    // Construct the linked list and read in records.

    fp = _Ropen(" MY_LIB/MY_FILE", "rr, blkrcd = Y");
    start = (struct link *) malloc(sizeof(struct link));
    start->next = NULL;
    ptr = start;
    for ( i = (_Ropnfbk(fp))>num_records; i > 0; --i )
    {
        _Rreadn(fp, NULL, MAX_LEN, __DFT);
        (struct link *) malloc(sizeof(struct link));
        memcpy(ptr->record, (void const *) *(fp->in_buf), MAX_LEN);
        ptr->last = 0;
    }
    ptr->last = 1;
    ptr = start->next;
    free(start);
    start = ptr;
    _Rclose(fp);

    // Process all records.

    if ( start != NULL )
    {
        for ( ptr = start; !ptr->last; ptr = ptr->next )
        {
            // Code to process the element pointed to by

```

```

    }
    // code to process the element
    //(last element) pointed.
    // Free space allocated for the linked list.

    while ( !start->last )
    {
        ptr = start->next;
        free(start);
        start = ptr;
    }
    free(start);
}
}

```

Reduce Indirect Access

You can improve performance by reducing indirect access through pointers. Each level of indirection adds some overhead:

```

for ( i = 0; i < n; i++ )
{
    x->y->z[i] = i;
}

```

Performance in the above example improves if it is rewritten as:

```

temp = x->y;
for ( i = 0; i < n; i++ )
{
    temp->z[i] = i;
}

```

Shallow Copy and Deep Copy

Avoid performing a *deep copy* if a *shallow copy* is all you require. For an object that contains pointers to other objects, a shallow copy copies only the pointers and not the objects to which they point. The result is two objects that point to the same contained object. A deep copy copies the pointers and the objects they point to, as well as any pointers or objects contained within that object, and so on.

Note: Care must be taken that objects which use shallow copy do not destroy objects pointed to more than once.

Space Considerations

You can improve the performance of a program by reducing the space it requires. Reducing the space requirement helps reduce page faults, segment faults, and effective address overflows.

Choose Appropriate Data Types

Choosing the appropriate data type can help improve your program's performance. If possible use short instead of int, and float instead of double. C++ uses 2 bytes for short, 4 bytes for int, and 8 bytes for double. Using the right data types can reduce your program's space requirement.

Note: When choosing data types, consider all the platforms that your code must support. You may not know all the data types and sizes at the beginning of your code design. Since the data types can hold the same size data on various platforms, you can use typedefs, enums, or classes depending on the use of the data type.

Reduce Dynamic Memory Allocation Calls

You can improve performance by reducing the number of times you dynamically allocate memory. Every time you call the `new` operator a certain amount of space is allocated from the heap. This space is always aligned at 16 bytes, which is suitable for storage of any object type. In addition, 32 extra bytes are taken from the dynamic heap for bookkeeping. This means that even if you only want one byte, 48 bytes are allocated from the dynamic heap, 32 bytes for bookkeeping and 15 bytes for padding. When the current space allocation in the heap is used up, storage allocation is slower:

```
ptr1 = new char[12];  
ptr2 = new char[4];
```

In the code above, 96 bytes are taken from the heap (including 64 bytes for bookkeeping and 16 bytes for padding) and `new` is used twice. This code can be rewritten as:

```
ptr1 = new char[16];  
ptr2 = ptr1 + 12;
```

Only 48 bytes are taken from the heap and the `new` operator is only used once. Since you reduce the dynamic space allocation requirement, less storage is taken from the heap. You may gain other benefits such as a reduction in page faults. Since there are fewer calls to the `new` operator, function call overhead is reduced as well.

Note: If allocating by incrementing pointers you must guarantee the proper alignment when allocating pointers or aggregates which can contain pointers (16 byte alignment). There is a performance degradation for types such as `float` whose natural alignment is word or doubleword if not allocated on their natural boundary.

Reduce Space Used for Padding

Reducing space wasted on padding by rearranging variables is another way to reduce your program's space requirement. In the C++ language:

- A `char` type variable takes one byte
- A `short` type variable takes 2 bytes
- An `int` type variable takes 4 bytes
- A `float` type variable takes 4 bytes
- A `double` type variable takes 8 bytes
- A pointer takes 16 bytes
- A `_DecimalT` template class object takes 1 to 16 bytes

Each data type is aligned on its natural boundary. pointers are aligned on 16 byte boundaries, integers are aligned on 4 byte boundaries and packed decimals are aligned on a 1 byte boundary. The alignment of a structure or a union is determined by the largest alignment among its members. This class aligns on a 4 byte boundary because of the `float` member.

Note: To support portable code, you can place all the members of the same type together (this may affect readability) starting with the largest (on most systems) going to the smallest. An array has the same alignment as its element type and must be positioned appropriately. This program that follows shows this rule implicitly.

```

class OrderT
{
    float value;    // Four bytes.
    char flag1;    // One byte plus one byte.
    short num;    // Two bytes.
    char flag2;    // One byte plus three bytes.
}
orderT;

```

The size of the above structure is 12 bytes (not 8 bytes). This is because there is 1 byte of padding after member `flag1` since member `num` aligns on a 2 byte boundary, and there are 3 bytes of padding after member `flag2` since the structure is aligned on a 4 byte boundary.

By rearranging variables, the wasted space created by padding can be minimized. Consider the structure as:

```

class ItemT
{
    char *name;                // 16 bytes.
    int number;                // 4 bytes plus 12 bytes.
    char *address;            // 16 bytes.
    double value;              // 8 bytes plus 8 bytes.
    char *next;                // 16 bytes.
    short rating;              // 2 bytes plus 14 bytes.
    char *previous;            // 16 bytes.
    _DecimalT<25,5> tot_order; // 13 bytes plus 3 bytes.
    int quantity;              // 4 bytes.
    _DecimalT<12,5> unit_price; // 7 bytes plus 5 bytes.
    char *title;               // 16 bytes.
    char flag;                 // 1 byte plus 15 bytes.
}
itemT;

```

The structure takes 176 bytes, of which 57 bytes are used for padding. It can be rearranged as:

```

class ItemT
{
    char *name;                // 16 bytes
    char *address;            // 16 bytes
    char *next;                // 16 bytes
    char *previous;            // 16 bytes
    char *title;               // 16 bytes
    double value;              // 8 bytes
    int quantity;              // 4 bytes
    int number;                // 4 bytes
    short rating;              // 2 bytes
    char flag;                 // 1 byte
    _DecimalT<25,5> tot_order; // 13 bytes
    _DecimalT<12,5> unit_price; // 7 bytes plus 9 bytes
}
itemT;

```

The class only takes 128 bytes, with 9 bytes for padding. The saving of space is even more substantial when you have arrays of the above structure type.

As a general rule, the space used for padding can be minimized if 16 byte variables are declared first, 8 byte variables are declared second, 4 byte variables are declared third, 2 byte variables are declared fourth, and 1 byte variables are declared fifth. `_DecimalT` template class objects should be declared last, after all other variables have been declared. The same rule can be applied to structure or class definitions.

Note: The /LB+ compiler option along with /LS+ shows the layout, including padding, of the structures in a module, in both `_Packed` and normal alignment.

Remove Observability

A module has **observability** when it contains data that allows it to be changed without being compiled again. Two types of data can make a module observable:

- Create Data** This data is necessary to translate the code into machine instructions. The object must have this data before you can change the optimization level. It is represented by the `*CRTDTA` value on the `RMVOBS` parameter of the Change Program (CHGPGM) command.
- Debug Data** This data enables an object to be debugged. It is represented by the `*DBGDTA` value on the `RMVOBS` parameter of the CHGPGM command.

The addition of these types of data increases the size of the object. Consequently, you may at some point want to remove the data in order to reduce object size. However, once the data is removed, so is the object's observability. To regain it, you must recompile the source and re-create the program.

To remove either kind of data from a program or module, use the CHGMOD or the CHGPGM command. Again, once you have removed the data, it is not possible to change the object in any way unless you re-create it. Therefore, ensure that you have access to all source required to create the program, or that you have a comparable program object with create data.

Compress an Object

The Create Data (`*CRTDTA`) value associated with an ILE program or module may make up more than half of the object's size. By removing or compressing this data, you reduce the secondary storage requirements for your programs significantly.

An alternative is to compress the object through using the Compress Object (CPROBJ) command. A compressed object takes up less system storage than an uncompressed one. When the compressed program is called, the part of the object containing the executable code is automatically decompressed. You can also decompress an object by using the Decompress Object (DCPOBJ) command.

Activation Groups

Using activation groups can impact performance.

Calling Functions in Other Activation Groups

Within the same job, calling a function that runs in a different activation group degrades the performance of the call significantly (the call takes approximately two times longer).

If a service program was created to run in a named activation group (using the `ACTGRP(name)` parameter of the CRTSRVPGM command) then any calls to that function from a program or service program would be calling *across an activation group* and would therefore be slower. Sometimes it makes sense to run programs or service programs in other activations groups (for storage isolation, exception handling) but it should be noted that call-performance suffers in that arrangement.

Reducing Program Start-Up Time

When a new C++ program is first called, the system needs to perform some initialization to prepare the program to run. Part of this initialization requires creating an activation group for all of the program storage, resolving all service programs bound to the program, getting program arguments and so on. Several recommendations for improving start-up time can be drawn from these initialization steps:

- Reduce the use of global variables.
- Reduce the number of service programs bound to the program. The more service programs used by an ILE program, the more time is required to start up the program. It is often better to have fewer, larger service programs than many, smaller ones. The C run-time is made up of a small number of service programs.

Program Control

Virtual functions and operators can impact performance.

Avoid Virtual Functions

There is a performance impact if you use virtual functions because virtual functions are compiled to be indirect calls, which are slower than direct calls. You may be able to minimize this performance impact depending on your program design by using a minimum number of parameters on the virtual functions.

Use Cheaper Operators

The C++ language has many features that other languages do not provide which can be used to improve your C++ program's performance:

- Instead of dividing an integer by 2^n , shift the integer right by n bits
- Instead of using the remainder operator for number 2^n ($x \% 32;$), use the bitwise-and operator & `0x0000001F`;

Compile-Time Performance Tips

There are several ways to improved compile-time performance. These include both front and back end compile-time activities.

Choosing Compiler Options

Table 32 on page 439 describes different compiler options to make your program run faster, and to make your compile program smaller. Note that sometimes you have to decide which is more important to you, program size or program speed. In some cases optimizing for one aspect means the other suffers.

Optimization is the process through which the system looks for processing shortcuts that reduce the amount of system resources necessary to produce output. Processing shortcuts are translated into machine code, allowing the procedures in a module to run more efficiently. A highly optimized program or service program should run faster than it would without optimization.

You control the level of optimization through the OPTIMIZE option on the Create Module and Create Bound Program commands. Changing the desired optimization level requires recompiling your source code. Changing the optimization of a module can also be accomplished through a CHGMOD command. Note that if the optimization level is 10, you can not change the optimization level without recompiling your source code.

You should be aware of the following limitations when working with optimized code:

- In general, the higher the optimizing request, the longer it takes to create an object.
- At higher levels of optimization, the values of fields may not be accurate when they are displayed in a debug session, or after the program recovers from an exception.
- Optimized code may have altered breakpoints and step locations used by the source debugger, since the optimization changes may rearrange or eliminate some statements.

To circumvent this restriction while debugging, you can lower the optimization level of a module to display fields accurately as you debug a program, and then raise the level again afterwards, to improve the program efficiency as you get the program ready for production.

Use the guidelines in Table 32, except where they are contradicted. Intrinsic functions may improve performance, but they increase the size of your module. Unless noted, these options are not set by default.

Table 32. Compiler Options for Performance

Option	Optimize for Speed	Optimize for Size
OPTIMIZE 10 OPTIMIZE 20 OPTIMIZE 30 OPTIMIZE 40 Turns on optimization.	Yes	Yes
INLINE(*OFF) Does not inline user functions. Not inlining may reduce module size especially if the inlined functions consist of small pieces of code.	No	Yes
INLINE(*ON) Inlines user functions.	Yes	No
DBGVIEW(*NONE) Does not generate debug information, which would increase module size.	No	Yes

Run-Time Limits

- The maximum amount of storage of any single variable (such as a string or array) is 16 773 104 bytes
- The maximum length of a command passed to the system function is 32 702 bytes
- The maximum size of dynamic heap storage is 4 gigabytes

A very large memory allocation may cause a system crash if there is insufficient auxiliary storage on your system. A 4 gigabyte memory allocation requires more than 4 gigabytes of available DASD. The iSeries Work System Status (WRKSYSSTS) command shows auxiliary storage usage.

- The maximum size of a single heap allocation is 16 711 568 bytes
- The maximum auto storage is 16MB and there is a recursion limit of around 21743 levels deep

Appendix B. Support for Data Description Specifications (DDS)

By using the GENCSRC utility to generate database header files, you can extract DDS information and create a header file with declarations for use in your programs.

GENCSRC provides the means to retrieve externally- described file information for use in a C/C++ program. The command creates a C/C++ header file which contains the typedef structure for the include file. The GENCSRC command lets you create C/C++ include files. #pragma mapinc exploits GENCSRC internally, and provides the opportunity to make DDS => include_file conversion directly with the help of the GENCSRC command. GENCSRC can produce include files in IFS file system, while #pragma mapinc produces headers only in database file system; GENCSRC include files can be placed permanently anywhere, while #pragma mapinc writes generated include files into QTEMP library.

The following table shows the comparison of #pragma mapinc options and the new keywords for GENCSRC. For more information on any particular option, refer to the description of #pragma mapinc in the *VisualAge for C++ for AS/400 C++ Language Reference* manual.

Keyword	#pragma mapinc option	Description
SRCFILE	member_name	The name of the file that you reference on the #include directive in the source program. The output file is generated in the Data Management file system.
SRCMBR	member_name	The name of member with the header information. It follows the iSeries naming conventions. The output file is generated in the Data Management file system.
OBJ	file_name	The path name of the object to map in QSYS file system.
SRCSTMF	member_name	The output file is generated in the IFS file system.
RCDFMT	format_name	Indicates the DDS record format to be included in your program. The default is *ALL.
SLTFLD	options	Restricted to a combination of the following values: <ul style="list-style-type: none">• *INPUT• *OUTPUT• *BOTH (default)• *KEY• *INDICATOR• *LVLCHK• *NULLFLDS
PKDDECFLD	d or p	*DECIMAL or *CHAR
STRUCTURES	_P	*NONPACKED or *PACKED
ONEBYTE	1BYTE_CHAR	*CHAR or *ARRAY
UNIONDFN	union_name	*OBJ or *NONE
TYPEDEFPIX	prefix_name	*OBJ or *NONE

Appendix C. Porting Programs to ILE C++

This section describes the differences between ILE C and ILE C++. Some examples are provided to illustrate how to port code.

Note: If you are developing new code, then follow the ANSI guidelines as much as possible and avoid platform-specific extensions. In general, your code should be portable. Platform specific extensions, for example, `_Far16`, `_Pascal16` are platform-specific pointers that are not portable.

Differences Between C and C++

This section describes the differences between C and C++.

Inter-language Calls

ILE C source code containing inter-language calls must be modified to run under ILE C++. The extern linkage specification with the function definition or declaration must be used instead of the **#pragma linkage** or **#pragma argument** directives. The **#pragma map** directive has some semantic differences.

There is a difference in the way the **#pragma argument** directive and the extern linkage specification handle function definitions. Both generate the same code when processing a function call but the **#pragma argument** directive does not affect parameters within the function definition. The extern linkage specification does affect parameters within the function definition.

These two modules in ILE C

Module1.C

```
extern void foo (int *i, char **s)
{
    *s = *i ? "Not Zero" : "Zero";
}
```

Module2.C

```
extern void foo (int, char *)

#pragma argument (foo, VREF)

int main()
{
    char *s;
    foo (1, s);
}
```

are equivalent to this ILE C++ code:

```
extern "VREF" void foo (int i, char *s)
{
    s = i ? "Not Zero" : "Zero";
}

int main()
```

```

{
    char *s;
    foo (1, s);
}

```

This code shows how `extern "OS"` with a function definition is used to replace the **#pragma linkage** directive. See Chapter 16, “Calling Conventions” on page 325 for additional information. Instead of using

```

#pragma datamodel (p128)
typedef void (FUNC)(int);
#pragma linkage (FUNC, OS)
#pragma datamodel (pop)

```

you should use

```
extern "OS" typedef void (FUNC) (int); // works
```

not

```

typedef void (FUNC)(int)
extern "OS" FUNC;           //error

```

Binary Coded Decimal Class Library for OS/400

The Binary Coded Decimal Class Library for OS/400 is provided so that you can create binary coded decimal objects that are compatible with the ILE **packed decimal** data types. To move the ILE C packed decimal data type to the `_DecimalT` class template you need to consider some differences between the two.

Macros Defined in `bcd.h`

These macros are used by the `_DecimalT` class template to maintain compatibility with ILE C:

```

#define decimal      _Decimal
#define digitsof    __digitsof
#define precisionof __precisionof
#define _Decimal(n,p) _DecimalT<n,p>
#define __digitsof(DecName) (DecName).DigitsOf()
#define __precisionof(DecName) (DecName).PrecisionOf()

```

This program shows the source to code a packed decimal data type in ILE C:

```

// ILE C program

#include <decimal.h>

void main()
{
    int dig, prec;
    decimal(9,3) d93;
    dig = digitsof(d93);
    prec = precisionof(d93);
}

```

This program shows the same source written in C++:

```

// C++ program

#include <bcd.h>

void main()
{
    int dig, prec;

```



```

    _DecimalT<9,3> d93;
    dig = d93.DigitsOf();
    prec = d93.PrecisionOf();
}

```

This program shows you that by using the macros defined in <bcd.h>, you can use the same ILE C shown in the first program:

// C++ program using the macro

```

#include <decimal.h>

void main()
{
    int dig, prec;
    decimal(9,3) d93;
    dig = digitsof(d93);
    prec = precisionof(d93);
}

```

Note: The <decimal.h> header file is specified since <decimal.h> includes the <bcd.h> header file.

Member of a Union

Since an object of a class with a constructor cannot be a member of a union, the `_DecimalT` class template in ILE C++ cannot be used as a member of a union.

Member of a Structure

In ILE C++ if a `_DecimalT` class template is a member of a struct, that struct cannot be initialized with an initializer list. The structure in ILE C is:

```

typedef struct {
    char      s1;
    decimal(5,3) s2;
}s_type;

s_type s ={'+', 12.345d};

```

In ILE C++ you need to rewrite the code as follows:

```

struct s_type {
    char s1;
    decimal(5,3) s2;
    s_type(char c, decimal(5,3) d) : s1(c), s2(d) {}
};

s_type s ('+', __D("12.345"));

```

Decimal Constant

The decimal constant defined using the suffix D or d is not supported by the C++ `_DecimalT` class template. Instead, a string literal embraced by `__D` is used to represent a packed decimal constant. The decimal constant 123.456D defined in ILE C equivalent to `__D("123.456")` in ILE C++.

Decimal Constant and Case Statements

The `__D` macro is used to simplify code that requires the frequent use of the `_ConvertDecimal` constructor. Because the `__D` macro is equivalent to the `_ConvertDecimal` constructor, the `__D` macro cannot be used with a case statement. A valid case statement uses an integral constant expression. This code results in a compiler error:

```

decimal(4,3) op;

switch int(op) {

```

```

        case int(__D("1.3")):
            .....
            break;
    }

```

The compiler flags the case statement indicating that the case expression is not an integral constant expression.

_DecimalT Class Template Specifiers

C++ uses these macros:

```

#define decimal      _Decimal
#define _Decimal(n,p) _DecimalT<n,p>

```

to map the template class instantiation to the desired ILE C syntax. ILE C code using the decimal(n,p) specifier can be ported to C++ without any modification. The second type specifier supported by ILE C is not supported by the C++ compiler:

►—decimal—(*constant-expression*)—————►

In this case, you need to insert a zero explicitly at the type specifier:

change decimal(10) to decimal(10,0)

Error Checking

In ILE C packed decimal is implemented as a native data type. This allows an error such as invalid decimal format to be detected at compile time. In C++ detection of a similar error is deferred until run time:

```

#define _DEBUG 1

#include <bcd.h>

void main()
{
    _DecimalT<10,20> b= __D("ABC"); // Run-time exception is raised
}

```

and

```

#define _DEBUG 1

#include <bcd.h>

void main()
{
    _DecimalT<33,2> a;           // Max. dig. allow is 31. Again,
                                // run-time exception is raised
}

```

Some errors can occur at compile time. When n<1, (_Decimal<-33,2>) then the error is detected at compile time.

Mathematical Operators: ILE C provides additional error checking on the sign or the digit codes of the packed decimal operand. Valid signs are hex A-F and the valid digit range is hex 0-9. If the decimal operand is in error, ILE C generates an error message. This additional checking is not present in the _DecimalT template class. This code results in an error message in ILE C but not in ILE C++.

```

#include <decimal.h>

void main()

```

```

{
    _Decimal(10,2) a, b;
    int c;
    c = a > b;    // a and b are not valid packed decimals because
                  // a and b are not initialized
}

```

Extra Precision

The `_DecimalT` class template allows a maximum of 62 and 93 digits as the internal results for the multiplication and division operations respectively. This is different from the ILE C packed decimal data type in which a maximum of 31 digits is used for both operations.

This internal result is different from the intermediate result. The internal result is designated to store the temporary result during the operation. After the operation is completed, the internal result is converted to the intermediate result and returned to the caller.

Header File

In ILE C, the header file `<decimal.h>` must be included in the source prior any usage of the packed decimal data type. In ILE C++ `<bcd.h>` must be included instead.

Digits of an Object

In ILE C, when you use the `__digitsof` operator with a packed decimal data type the result is an integer constant. The `__digitsof` operand can be applied to a packed decimal data type or a packed decimal constant expression. The `__digitsof` operator returns the total number of digits n in a packed decimal data type.

To determine the number of digits n in a packed decimal data type:

```

#include <decimal.h>

int n,n1;
decimal (5, 2) x;

n = __digitsof(x);          /* the result is n=5 */
n1 = __digitsof(1234.567d); /* the result is n1=7 */

```

In ILE C++ when you use the member function `DigitsOf()` with a `_DecimalT` class template the result is an integer constant. The member function `DigitsOf()` can be applied to a `_DecimalT` template class object. The member function `DigitsOf()` returns the total number of digits n in a `_DecimalT` template class object.

To determine the number of digits n in a `_DecimalT` template class object:

```

#include <bcd.h>

int n,n1;
_DecimalT <5, 2> x;

n = x.DigitsOf();           // the result is n=5

```

Precision of an Object

When you use the `__precisionof` operator with a packed decimal data type the result is an integer constant. The `__precisionof` operand can be applied to a packed decimal data type or a packed decimal constant expression. The `__precisionof` operator tells you the number of decimal digits p of the packed decimal data type.

To determine the number of decimal digits p of the packed decimal data:

```
#include <decimal.h>

int p,p1;
decimal (5, 2) x;

p=__precisionof(x);          /* The result is p=2 */
p1=__precisionof(123.456d);  /* The result is p1=3 */
```

In ILE C++ when you use the member function `PrecisionOf()` with a `_DecimalT` class template the result is an integer constant. The member function `PrecisionOf()` can be applied to a `_DecimalT` template class object. The member function `PrecisionOf()` tells you the number of decimal digits *p* of the `_DecimalT` template class object.

To determine the number of decimal digits *p* of the `_DecimalT` class template object:

```
#include <bcd.h>

int p,p1;
_DecimalT <5, 2> x;

p=x.PrecisionOf();          // The result is p=2
```

Using Formatted C Input and Output Functions

The behavior of the `fprintf()`, `sprintf()`, `vfprintf()`, `vprintf()` and `vsprintf()` functions is the same as the `printf()` function. The behavior of the `fscanf()` and `sscanf()` functions is the same as the `scanf()` function.

To control the format of the printout use the *flags*, *width* and *precision* fields of the `printf()` function.

To control the format of the `scanf()` function use the *** and *width* fields of the `scanf()` function. See the *VisualAge for C++ for AS/400 C Library Reference* for information on these fields.

Print Function Flags: Table 33 describes the flag characters and their meanings for `D(n,p)` conversions.

Table 33. Flag Meanings for Printing the Value of a `_DecimalT` Template Class Object

Flag	Meaning
#	The result always contains a decimal-point character, even if no digits follow it.
0	Leading zeros (following any indication of sign or base) are used to pad to the field width; no space padding is performed.
-	The result is always left-justified within the field.
+	The result always begins with a plus or minus sign.
space	The result is always prefixed with a space where the result of a signed conversion is no sign or the signed conversion results in no characters.

Print Function Field Width: The optional minimum field width for the `printf()` function indicates the minimum number of digits to appear in the integral part, fractional part or both parts of a `_DecimalT` template class object. If there are fewer characters than the field width, then the field is padded with spaces. The field width can be an ***. If *n* is an ***, the value of *n* is derived from the corresponding position in the parameter list.

Print Function Field Precision: The optional precision for the `printf()` function indicates the number of digits to appear in the fractional part of a `_DecimalT`

template class object. The default precision is *p*. If precision is greater than *p*, extra zeros are padded. If precision is less than *p*, rounding is performed. The field precision can be an ***. If *p* is an ***, the value of *p* is derived from the corresponding position in the parameter list.

Conversion Specifiers: The conversion specifier for the `printf()` function is:

D(n,p) The `_DecimalT` template class object is converted in the style `[-] ddd.ddd` where the number of digits after the decimal-point character is equal to the precision of the specification. If the precision is missing, it is taken as 0; if the precision is zero and the `#flag` is not specified, no decimal-point character appears. If a decimal-point character appears, at least one digit appears before it. The value is truncated to the appropriate number of digits. The `(n,p)` descriptor is used to describe the characteristic of the `_DecimalT` template class object. Both *n* and *p* have to be in the form of decimal integers. If *p* is missing, a default value of zero is assumed. If the specifier is in another form not stated above, the behavior is undefined.

If *n* and *p* of the variable to be printed do not match with the *n* and *p* in the conversion specifier `%D(n,p)`, the behavior is undefined. Use the unary operators `__digitsof (expression)` and `__precisionof (expression)` in the argument list to replace the *** in `D(*,*)` whenever the size of the resulting class of a `_DecimalT` template class object expression is not known.

The conversion specifier for the `scanf()` function is as follows:

D(n,p) Matches a decimal number, the expected form of the subject sequence is an optional plus or minus sign, then a nonempty sequence of digits optionally containing a decimal point. The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign, a digit, or a decimal-point character.

Conditional Operator

In C++ both the second and third expressions must be of the same class. In ILE C if both the second and third operands have an arithmetic type, the usual arithmetic conversions are performed to bring them to a common type. In C++ if either expression has a different class, then you must cast the second or third expression so that it has the same class. The following illustrates an example where the conditional expression fails because the second and third expressions are not of the same class.

```
#include <bcd.h>

main()
{
    int var_1;
    decimal(4,2) op_1_1 = _D("12.34");
    decimal(10,2) op_1_2 = _D("123.45");
    var_1 = (op_1_1 < op_1_2) ? (op_1_1 + 3) : op_1_2;
}
```

To use the conditional operator with the `_DecimalT` template class you can do one of the following:

- Use an explicit cast on the second expression so that it has the same type as the third expression
- Use the same type of variables

This program shows an explicit cast on the second expression so that it has the same class as the third expression:

```
#include <bcd.h>

main()
{
    int          var_1;
    decimal(4,2) op_1_1 = __D("12.34");
    decimal(10,2) op_1_2 = __D("123.45");
    var_1 = (op_1_1 < op_1_2) ? (_DecimalT<10,2>)__D(op_1_1 + 3) : op_1_2;
}
```

This program shows how to use the same type of variables:

```
#include <bcd.h>

main()
{
    int var_1;
    decimal(10,2) op_1_1 = __D("12.34");
    decimal(10,2) op_1_2 = __D("123.45");
    var_1 = (op_1_1 < op_1_2) ? op_1_1 : op_1_2;
}
```

Note: The + 3 was removed from the second expression since (op_1_1 + 3) results in _Decimal<13,2>.

Porting ILE C Packed Decimal Data Type to _DecimalT Class Template

In the class template _DecimalT, neither the constructor nor the assignment operator are overloaded to take any of the template class instantiated from _DecimalT. For this reason, explicit type casting that involves conversion from one _DecimalT template class object to another cannot be done directly. Instead, the macro __D must be used to embrace the expression that requires explicit type casting. This program is written in ILE C:

```
#include <decimal.h>

void main ()
{
    decimal(4,0) d40 = 123D;
    decimal(6,0) d60 = d40;
    d60 = d40;
    decimal(8,0) d80 = (decimal(7,0))1;
    decimal(9,0) d90;
    d60 = (decimal(7,0))12D;
    d60 = (decimal(4,0))d80;
    d60 = (decimal(4,0))(d80 + 1);
    d60 = (decimal(4,0))(d80 + (float)4.500);
}
```

This source needs to be rewritten as:

```
#include<bcd.h>

void main ()
{
    _DecimalT<4,0> d40 = __D("123"); // OK
    _DecimalT<6,0> d60 = __D(d40);    // Because no constructor
                                     // exists that can convert d40 to d60.
                                     // macro __D is needed to convert d40
                                     // into an intermediate type first.
    d60 = d40;                        // OK. This is different from the
                                     // previous statement in which
                                     // the constructor was called.
```

```

// In this case, the assignment
// operator is called and the
// compiler converts d40 into the
// intermediate type automatically.
_DecimalT<8,0> d80 = (_DecimalT<7,0>)1;
// OK
// Type casting an int, not a decimal(n,p)

_DecimalT<9,0> d90; // OK
d60 = (_DecimalT<7,0>)__D("12"); // OK
d60 = (_DecimalT<4,0>)__D(d80);
d60 = (_DecimalT<4,0>)__D(d80 + 1);

// In both cases, the resultant classes
// of the expressions are _DecimalT<n,p>.
// macro __D is needed to convert them
// to an intermediate type first.

d60 = (_DecimalT<4,0>)(d80 + (float)4.500);
// OK because the resultant type
// is a float
}

```

Header Files that Work with Both C and C++

C header files are not generally usable by C++. Structures, unions and type definitions may be all right as well as variables. Care must be used in function prototypes and pragmas.

C/C++ Enablement

Wrap your header files in the following construct:

```

.
.
.
#ifdef __cplusplus
    extern "C" {
        #pragma info(none)
    #else
        //only if you have #pragma
        #pragma nomargins nosequence //nomargin and #pragma checkout in the
        #pragma checkout(suspend)    //header file
    #endif
    .
    .
    .
#ifdef __cplusplus
    #pragma info(restored)
    {
    #else
        #pragma checkout(resume)
    #endif
    }
}

```

The linkage specification `extern "C"` informs the compiler that all functions prototyped will have C linkage. C++ linkage functions cannot be called from C using the C++ internal name. See Chapter 16, "Calling Conventions" on page 325 for information on calling functions from other languages.

The macro `__ILEC400__` can replace `__cplusplus` but `__cplusplus` is preferred because it is portable to other implementations.

Packed Structures

The `_Packed` keyword tells the compiler to ignore the padding and pack the structure as much as possible. In ILE C this keyword can be used in a structure

definition and typedef. The same keyword can only be used in a typedef in the C++ compiler.

Table 34. Comparing Packed Structures

	ILE C/400	ILE C/C++
typedef _Packed struct { . }ps_t;	ok	ok
_Packed struct { . }ps_v;	ok	error

Therefore, you must make sure the **_Packed** keyword is only used in typedefs in the header file.

In the ILE C/C++ compiler, the **#pragma pack** directive applies only to C programs. The ILE C **#pragma pack** directive is not compatible with the Windows® **#pragma pack** directive.

Function Prototype

To allow your header files to be used by ILE C and ILE C++ compilers, all functions with "OS" linkage type must be dual prototyped:

```
#ifdef __cplusplus
    extern "linkage-type" //linkage type "OS"
#else
    #pragma linkage(function_name,linkage_type)
#endif
void function_name(...);
```

If you have a list of functions that need dual prototypes, you can:

```
#ifdef __cplusplus
    extern "linkage-type" { //linkage type "OS"
#else
    #pragma linkage(function_name1, linkage_type)
    .
    .
    #pragma linkage(function_nameN, linkage_type)
#endif
void function_name1(...);
.
.
void function_nameN(...);
#ifdef __cplusplus
}
#endif
```

Enum Data Type Size

In ILE C++ the enum size is always the size of an integer unless the **#pragma enumsize** is used. If the size of the enum type is critical, the following code can be used to resolve the problem:

```
.
.
#pragma enumsize (2)
enum { a=0xffff} A; //sizeof(A)=2;
#pragma enumsize ()
.
.
```


Including QSYSINC Header Files

To use the QSYSINC header files in ILE C++ you need to use the following convention `#include <file/header.h>`. To include QSYSINC/MIH/SYSEPT you can use `#include <mih/sysept.h>`

This form also works for ILE C.

Type Checking

Type checking is stricter in C++ than it is in C. C++ allows void pointers to be assigned only to other void pointers. In ANSI C, a pointer to void can be assigned to a pointer of any other type without an explicit cast. See the *VisualAge for C++ for AS/400 C++ Language Reference* for information on compatibility.

Assume some source uses `memcmp()` to compare a constant char array to a volatile char array. The C compiler compiles this code without errors. Compiling the same code with the C++ compiler will result in an error message, for example, `Volatile unsigned char` cannot be converted to a `const void` pointer. You cannot use a constant pointer where a volatile pointer is expected unless you cast it. You can cast a void pointer to the appropriate pointer type before compiling the code.

C memory functions such as `malloc()`, `calloc()`, `realloc()`, that return void pointers must be cast to an appropriate pointer type before compiling the code. You can use the `new` and `delete` operators instead of `malloc()` and `free()` to take advantage of C++.

Name Mangling

All C++ function names are mangled to enable function overloading. You receive an undefined names error when you bind ILE C/C++ functions with mangled names, for example, `LocateSpaces__FPc`. In ILE C, the service program relationship is `LocateSpaces__FPc == LocateSpaces` or `LocateSpace__FPc == LocateSpace`

If you are porting ILE C code and you want to stop function name mangling then use `extern "C"` around the function name.

File Inclusion

The include file name must be a valid workstation file name, for example `"file_name"` or `<file_name>`. Include files cannot reference `*LIBL` or `*CURLIB` values. You can use these values in ILE C include names. For example, `("*LIBL/ABC", *LIBL/ABC/*A11)"...`.

If you use a back slash (`\`) character in your include file name, you must use a double back slash (`\\`). For example, `"c:\\abc\\aaa.h"` for a fully qualified path name or `"proj1\\aaa.h"` for a relative path include name. See the *VisualAge for C++ for AS/400 C++ Language Reference* for information on file inclusion.

Function Prototypes, Declarations and Pointers

C++ requires full prototype declarations. ANSI C allows non-prototyped functions.

In C++, all declarations of a function must match the unique definition of a function. ANSI C has no such restriction.

The type defined in a pointer declaration must match the type defined in the function prototype in C++. You can assign a pointer to a different type from what is defined in the function prototype by using an explicit cast expression:

```

void (*oldsig) (); // void pointer "void (*) ()"
extern "C" void (*signal (int, void(*) (int))) (int): // function
                                                    // prototype
                                                    // void(*) (int)
                                                    // returned

oldsig = signal (SIGALL, sig_handler);

```

CTT3055: "extern "C" void(*) (int) cannot be converted to "void (*) ()" results from the type mismatch between the type defined in void pointer ("void(*) ()") and the type defined in the function prototype (void (*) (int)). The *int* parameter does not exist in the function oldsig or the void pointer declaration.

In another example, QXX functions return unsigned char pointers. ILE C allows you to assign a signed char to an unsigned char pointer. This is not valid in C++. Unsigned char pointers must be declared as unsigned char variables in the source code as shown in the example source:

```

#include <xxcvt.h> //void(QXXITOP(unsigned char *pptr, int digits, int
                    //fraction, int value);

#include <stdio.h>

int main(void)
{
    unsigned char pptr[10];
    int digits = 3, fraction = 0;
    int value = 116;
    QXXITOP (pptr, digits, fraction, value);
}

```

You can assign the pointer to a variable of a different type by using a cast expression. You can also use the DFTCHAR compiler option when you compile the C++ source to set the default char type. See the *ILE C/C++ Compiler Reference* for more information on these options.

Character Array Initialization

In C++, when you initialize character arrays, a trailing `'\0'` (zero of type char) is appended to the string initializer. You cannot initialize a character array with more initializers than there are array elements.

In ANSI C, space for the trailing `'\0'` can be omitted in this type of information. The following initialization, for instance, is not valid in C++:

```

char v[3] = "asd"; //not valid in C++, valid in ANSI C
                    //because four elements are required

```

This initialization produces an error because there is no space for the implied trailing `'\0'` (zero of type char). The following initialization, for instance, is valid in C++:

```

char v[4] = "asd"; //valid in C++

```

See the *VisualAge for C++ for AS/400 C++ Language Reference* for information on compatibility.

String Literals

In ILE C strings are placed into read/write memory by default. In ILE C++ you must use the **#pragma strings** directive to place strings into read/write memory. The syntax of this pragma is:

►► #pragma strings ((readonly) writeable) ►►

C strings are read/write by default. C++ strings are read only by default. This pragma must appear before any C or C++ code in a file.

Integrated File System

The integrated file system provides a common interface to store and operate on information in stream files. The C stream I/O functions and the C++ stream I/O classes are implemented through the integrated file system. There are seven file systems in the integrated file system. The library (QSYS.LIB) file system maps to the iSeries file system but using this system under the integrated file system presents some limitations:

- Logical files are not supported
- The only types of physical files supported are program-described files containing a single field and source physical files containing a single text field
- Byte-range locking is not supported; see the *System API Reference*
- If any job has a database file member open, only one job is given write access to that file at any time; other jobs are allowed only read access

The C and C++ stream I/O default is integrated file system enabled using the ILE C++ for iSeries compiler. The ILE C compiler defaults to C data management stream I/O. If you have programs that use database or DDM files, your best choice is to use the SYSIFCOPT(*NOIFSIO) compiler option. This ensures that you compile your existing programs using the iSeries data management file system and not the integrated file system. Compiling programs that use restricted database or DDM files under the integrated file system results in a run-time error.

Set_Terminate is Scoped to an Activation Group

The effect of the `set_terminate()` function is scoped to an activation group. In the following example both `my_terminate()` and `a()` reside in a service program which runs in activation group "B". `main()` runs in another activation group, for example "A". In this scenario, the thrown exception by `a()` cannot percolate up to `main()`. This uncaught exception cannot cause `my_terminate()` to be invoked. As a result, CEE9901 is sent to `main()`.

```
// File main.c

#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <terminat.h>

void a();
void my_terminate();

int main() {
    set_terminate(my_terminate);
    try {
        a();
    }
    catch(...) cout << "failed" << endl;
}

// File term.c

#include <stdio.h>
```

```
#include <stdlib.h>
#include <iostream.h>
void my_terminate() {
    cout << "failed" << endl;
}
void a() { throw 7; }
```

But if the `set_terminate` statement is relocated in a prior to the `throw` operation, `my_terminate` can then be invoked. This is because the effect of `set_terminate` is scoped to an activation group. The activation group where `set_terminate` is executed is terminated (not the entire application, if the application encompasses several activation groups).

Appendix D. Using Templates in C++ Programs



Templates may be used in C++ to declare and define classes, functions, and static data members of template classes. The C++ language describes the syntax and meaning of each kind of template. Each compiler determines the mechanism that controls when and how often a template is expanded.

ILE C++ offers several alternative organizations with a range of convenience and compile performance to meet the needs of any program. This information describes those alternatives and the criteria you should use to select which one is right for you.

See the *VisualAge for C++ for AS/400 C++ Language Reference* for a general description of templates.

This section includes:

- “How the Compiler Expands Templates” on page 458
- “Generating Template Function Definitions” on page 459
- “Including Defining Templates” on page 460

Note: The *CURRENT and *PREV compilers handle templates differently. The *CURRENT compiler is compliant with ANSI 98, whereas the *PREV compiler follows the documentation for the previous release.

Using Template Terms

These terms describe the template constructs in C++:

class template

A template used to generate classes. Classes generated in this way are called *template classes*. A class template describes a family of related classes. It can simply be a declaration, or it can be a definition of a particular class.

function template

A template used to generate functions. Functions generated in this way are called *template functions*. A function template describes a family of related functions. It can be a declaration, or it can be a definition of a particular function.

declaring template

A class template or function template that includes a declaration but does not include a definition. A declaring function template is:

```
template<class A> void foo(A*a);
```

A declaring class template is:

```
template<class T> class C;
```

defining template

A class template or function template declaration that includes a definition. A defining function template is:

```
template<class A> void foo(A*a) {a ->Bar();};
```

A defining class template would look like this:

```
template<class T> class C : public A {public: void boo();};
```

explicit definition

A user-supplied definition that overrides a template. An explicit definition of the foo function is:

```
void foo(int *a) {a++;}
```

An explicit definition of a template class is:

```
class C<short> {  
    public: int moo();  
}
```

Instantiation

A defining template defines a whole family of classes or functions. An *instantiation* of a template class or function is a specific class or function that is based on the template.

How the Compiler Expands Templates

You can instantiate templates in three ways:

1. Include defining templates everywhere. See “Including Defining Templates Everywhere” on page 460 for more details.
2. Use the compiler’s automatic facility to ensure that there is a single instantiation of the template. See “Structuring for Automatic Instantiation” on page 460 for more details.
3. Manually structure your code so that there is a single instantiation of the template. See “Manually Structuring for Single Instantiation” on page 464 for more details.

The best way to instantiate templates depends on how the compiler reacts when it encounters templates.

When you use templates in your program, the ILE C++ compiler automatically instantiates each defining template that is:

- Referenced in the source code
- Visible to the compiler (included as the result of an #include statement)
- Not explicitly defined by the programmer

If an program consists of several separate compilation units that are compiled separately, a given template may expand in two or more of the compilation units. For templates that define classes, inline functions, or static nonmember functions, this is the desired behavior. These templates need to be defined in each compilation unit where they are used.

For other functions and for static data members, which have external linkage, defining them in more than one compilation unit would normally cause an error when the program is bound. ILE C++ avoids this problem by giving special treatment to template-generated versions of these objects. At bind time, ILE C++ gathers all template-generated functions and static-member definitions, plus any explicit definitions, and resolves references to them in these ways:

- If an explicit definition of the function or static member exists, it is used for all references. All template-generated definitions of that function or static member are discarded.

- If no explicit definition exists, one of the template-generated definitions is used for all references. Any other template-generated definitions of that function or static member are discarded.

You may have only one explicit definition of any external linkage template instance.

Generating Template Function Definitions

The class template `Stack` shows you template function generation. `Stack` implements a stack of items. The overloaded operators `<<` and `>>` are used to push items onto the stack and pop items from the stack. Assume that the declaration of the `Stack` class template is contained in the file `stack.h`:

```
template <class Item, int size> class Stack {
public:
    int operator << (Item item); // push operator
    int operator >> (Item& item); // pop operator
    Stack() { top = 0; } // constructor defined inline
private:
    Item stack[size]; // stack of items
    int top; // index to top of stack
};
```

In the template, the constructor function is defined inline. Assume the other functions are defined using separate function templates in the file `stack.c`:

```
template <class Item, int size>
int Stack<Item,size>::operator << (Item item) {
    if (top >= size) return 0;
    stack[top++] = item;
    return 1;
}
template <class Item, int size>
int Stack<Item,size>::operator >> (Item& item)
{
    if (top <= 0) return 0;
    item = stack[--top];
    return 1;
}
```

The constructor has internal linkage because it is defined inline in the class template declaration. In each compilation unit that uses an instance of the `Stack` class, the compiler generates the constructor function body. Each unit uses its own copy of the constructor.

In each compilation unit that includes the file `stack.c`, for any instance of the `Stack` class in that unit, the compiler generates definitions for the following functions (assuming there is no explicit definition):

```
Stack<item,size>::operator<<(item)
Stack<item,size>::operator>>(item&)
```

Given the source file `usrstack.cpp`:

```
#include "stack.h"
#include "stack.c"
void Swap(int i&, Stack<int,20>& s)
{
    int j;
    s >> j;
    s << i;
    i = j;
}
```

the compiler generates the functions `Stack<int,20>::operator<<(int)` and `Stack<int,20>::operator>>(int&)` because both those functions are used in the program, their defining templates are visible, and no explicit definitions are seen.

Including Defining Templates

There are three methods of including defining templates:

- “Including Defining Templates Everywhere”
- “Structuring for Automatic Instantiation”
- “Manually Structuring for Single Instantiation” on page 464

Automatic instantiation is the recommended method.

Including Defining Templates Everywhere

The simplest way to instantiate templates is to include the defining template in every compilation unit that uses the template. This alternative has these disadvantages:

- If you make even a trivial change to the implementation of a template, you must recompile every compilation unit that uses it.
- The compilation process is slower, and the resulting modules are bigger because the templates are expanded in every compilation unit where they are used. The duplicated code for the templates is eliminated during binding, so the executable programs are not larger if you choose to include defining templates everywhere.

To use this method with the `Stack` template, include both `stack.h` and `stack.c` in all compilation units that use an instance of the `Stack` class. The compiler generates definitions for each template function. Each template function may be defined multiple times, increasing the size of the module.

Structuring for Automatic Instantiation

The recommended way to instantiate templates is to structure your program for their automatic instantiation. The advantages of this method are:

- It is easy to do
- Unlike the method of including defining templates everywhere, you do not get larger modules and slower compile times
- Unlike the method of including defining templates everywhere, you do not have to recompile all of the compilation units that use a template if that template implementation is changed

The disadvantages of this method are:

- It may not be practical in a team programming environment because the compiler may update source files that are being modified at the same time by somebody else.
- The modifications that are made to source files may not be file-system-independent. Header files that are locally available may be included rather than header files that are available on a network.
- There are some situations where the compiler cannot determine exactly which header files should be included.

To use this facility:

1. Declare your template functions in a header file using class or function templates, but do not define them. Include the header file in your source code.

2. For each header file, create a *template-implementation* file with the same name as the header and the extension `.c`. Define the template functions in this template-implementation file.

Note: Use the same compiler options to bind your modules that you use to compile them.

```
CRTPGM PGM (MYLIB/MYPROG) MODULE(MYLIB/MYFILE)
```

This is especially important for options that control libraries, linkage, and code compatibility.

For each header file with template functions that need to be defined, the compiler generates a template-include file. The template-include file generates **#include** statements in that file for:

- The header file with the template declaration
- The corresponding template-implementation file
- Any other header files that declare types used in template parameters

Note: If you have other declarations that are used inside templates but are not template parameters, you must place or **#include** them in either the template-implementation file or one of the header files included as a result of the above three steps. Define any classes that are used in template arguments and that are required to generate the template function in the header file. If the class definitions require other header files, include them with the **#include** directive. The class definitions are then available in the template-implementation file when the function definition is compiled. Do not put the definitions of any classes used in template arguments in your source code.

```
foo.h
    template<class T> void foo(T*);
hoo.h
    void hoo(A*);
foo.c
    template<class T> void foo(T* t)
        {t -> goo(); hoo(t);}
other.h
    class A {public: void goo() {} };

main.cpp
    #include "foo.h"
    #include "other.h"
    #include "hoo.h"
    int main() { A a; foo(&a); }
```

This requires the expansion of the `foo(T*)` template with `class A` as the template type parameter. The compiler creates a template-include file `TEMPINC\foo.cpp`. The file contents (simplified below) are:

```
#include "foo.h"           //the template declaration header
#include "other.h"         //file defining template type parameter
#include "foo.c"           //corresponding template implementation

void foo(A*);              //triggers template instantiation
```

This does not compile properly because the header `"hoo.h"` did not satisfy the conditions for inclusion but the header file is required to compile the body of `foo(A*)`. One solution is to move the declaration of `hoo(A*)` into the `"other.h"` header file.

The function definitions in your template-implementation file can be explicit definitions, template definitions, or both. Any explicit definitions are used instead of the definitions generated by the template.

Before it invokes the binder, the compiler compiles the template-include files and generates the necessary template function definitions. Only one definition is generated for each template function.

The `TEMPLATE` parameter defaults to `*NONE`. The other options are `*SRCDIR` or `pathname`, where `pathname` is an IFS directory. The compiler creates the specified directory if it does not already exist. The applicable `xlC` qshell command option is `-qtempinc=dir`, where `dir` is a directory name. You can specify a fully qualified path name or a path name relative to the current directory.

If you specify a different directory for your template-include files, make sure that you specify it consistently for all compilations of your program, including the bind step.

Note: After the compiler creates a template-include file, it may add information to the file as each compilation unit is compiled. The compiler never removes information from the file. If you remove function instantiations or reorganize your program so that the template-include files become obsolete, you may want to delete one or more of these files and recompile your program. If error messages are generated for a file in the `TEMPINC` directory, you must either correct the errors manually or delete the file and recompile. To regenerate all of the template-include files, delete the `TEMPINC` directory, the modules, and recompile your program.

If you do not delete the modules, `MAKEFILE` rules prevent the modules from being recompiled, and the template-include files cannot be updated with all the lines needed for all the compilation units used in the program. The end result is that the bind fails.

A Template-Implementation File

In the Stack source, the file `stack.c` is a template-implementation file. To create a program using the Stack class template, `stack.h` and `stack.c` must reside in the same directory. You include `stack.h` in any source files that use an instance of the class. The `stack.c` file does not need to be included in any source files. Given the source file:

```
#include "stack.h"
void Swap(int i&, Stack<int,20>& s)
{
    int j;
    s >> j;
    s << i;
    i = j;
}
```

the compiler automatically generates the functions `Stack<int,20>::operator<<(int)` and `Stack<int,20>::operator>>(int&)`.

You can change the name of the template-implementation file or place it in a different directory using the **#pragma implementation** directive.

The syntax is:

The *path* is used to specify the path name for the template-implementation file. If it is only a partial path name, it must be relative to the directory containing the header file.

Note: This path is a quoted string following the normal conventions for writing string literals. Backslashes must be doubled.

In the Stack class, to use the file `stack.def` as the template-implementation file instead of `stack.c`, add the line: `#pragma implementation("stack.def")` anywhere in the `stack.h` file. The compiler then looks for the template-implementation file `stack.def` in the same directory as `stack.h`.

A Template-Include File

A typical template-include file generated by the compiler shows this information:

```
/*0000000000*/ #pragma sourcedir("c:\swearsee\src")
/*0698421265*/ #include "c:\swearsee\src\list.h"
/*0000000000*/ #include "c:\swearsee\src\list.c"
/*0698414046*/ #include "c:\swearsee\src\mytype.h"
/*0698414046*/ #include "c:\IBMCPP\INCLUDE\iostream.h"
#pragma define(List<MyType>)
ostream& operator<<(ostream&,List<MyType>);
#pragma undeclared
int count(List<MyType>);
```

0
1
2
3
4
5
6
7
8

- 0** This pragma ensures that the compiler looks for nested include files in the directory containing the original source file, as required by the ILE C++ file inclusion rules.
- 1** The header file that corresponds to the template-include file. The number in comments at the start of each `#include` line (for this line `/*0698421265*/`) is a time stamp for the included file. The compiler uses this number to determine if the template-include file is current or should be recompiled. A time stamp containing only zeroes (0) as in line **2** means the compiler is to ignore the time stamp.
- 2** The template-implementation file that corresponds to the header file in line **1**.
- 3** Another header file that the compiler requires to compile the template-include file. All other header files that the compiler needs to compile the template-include file are inserted at this point.
- 4** Another header file required by the compiler. It is referenced in the function declaration in line **6**.
- 5** The compiler inserts **#pragma define** directives to force the definition of template classes. In this case, the class `List<MyType>` is to be defined and its member functions are to be generated.
- 6** The `operator<<` function is a nonmember function that matched a template declaration in the `list.h` header file. The compiler inserts this declaration to force the generation of the function definition.
- 7** The **#pragma undeclared** directive is used only by the compiler and only in template-include files. All template functions that are explicitly declared in at least one compilation unit appear before this line. All template functions that are called, but never declared, appear after this line. This

division is necessary because the C++ rules for function overload resolution treat declared and undeclared template functions differently.

- 8** count is a template function that is called but not declared. The template declaration of the function is contained in `list.h`, but the instance `count(List<MyType>)` is never declared.

Note: Although you can edit the template-include files, it is not normally necessary or advisable to do so.

Manually Structuring for Single Instantiation

If you do not want to use the automatic instantiation method of generating template function definitions, you can structure your program in such a way that you define template functions directly in your compilation units. The advantage of this approach is that modules are smaller and compile times are shorter than they are when you include defining templates everywhere. When you structure your code manually for template instantiation, you avoid the potential problems that automatic instantiation can cause, such as dependency on a particular file system or file sharing problems.

There are disadvantages to structuring your code manually for template instantiation:

- You have to do more work than for the other two methods. You may have to reorganize source files and create new compilation units.
- You have to be aware of all of the instantiations of templates that are required by the entire program.

Note: It is recommended that you use the compiler's automatic instantiation facility. The manual structuring method is useful if you find you cannot work around the limitations of the automatic instantiation method.

Use **#pragma define** directives to force the compiler to generate the necessary definitions for all template classes used in other compilation units. Use explicit declarations of non-member template functions to force the compiler to generate them.

To use the second method, include `stack.h` in all compilation units that use an instance of the `Stack` class, but include `stack.c` in only one of the files. If you know what instances of the `Stack` class are being used in your program, you can define all of the instances in a single compilation unit:

```
#include "stack.h"
#include "stack.c"
#include "myclass.h" // Definition of "myClass" class
#pragma define(Stack<int,20>)
#pragma define(Stack<myClass,100>)
```

The **#pragma define** directive forces the definition of two instances of the `Stack` class without creating any object of the class. Because these instances reference the member functions of that class, template function definitions are generated for those functions. See the *VisualAge for C++ for AS/400 C++ Language Reference* for information about the `pragma` directive.

When you use these methods, you may also need to specify the `TEMPLATE(*NONE)` compiler option to suppress automatic creation of `TEMPINC` files. See *ILE C/C++ Compiler Reference* for more information about ILE C/C++ compiler options.

Appendix E. Casting with Run-Time Type Information



Run-Time Type Information (RTTI) is an extension to the C++ language made by the ANSI/ISO standard committee. You can classify extensions to the language as either minor (those extensions that simply provide a more convenient way of implementing traditional designs) or major (extensions that fundamentally affect the organization of programs). RTTI is a major extension — similar in impact to templates, exceptions and name spaces.

This section describes:

- “Introducing RTTI”
- “Using C++ Language Defined RTTI” on page 466
- “Understanding ILE C++ Extensions to RTTI” on page 469

Introducing RTTI

The RTTI language extension was designed to address the difficulty encountered by builders and users of major C++ libraries. The builders and users needed a mechanism for extending base classes provided in libraries. The RTTI mechanisms implemented by builders of the major C++ libraries were incompatible with each other. This meant it could be difficult to use more than one library for a given program, or to use the same program with different libraries without major changes to the program. Since the C++ language supports the reuse of code and the building of programs from parts, the incompatibilities of the RTTI mechanisms used internally by the various C++ libraries presented a challenge to the purpose of C++. A language-supported mechanism was needed.

There are three parts to RTTI support, each corresponding to increasing involvement and knowledge of the language’s RTTI implementation. The part of the mechanism that requires the least involvement and knowledge also serves the majority of needs. This is the part of the RTTI construct that you can focus on — **dynamic_cast**.

The parts of the C++ language support for RTTI are:

dynamic_cast operator

This operator combines type-checking and casting in one operation. It checks whether the requested cast is valid, and performs the cast only if it is valid.

typeid operator

This operator returns the run-time type of an object. If the operand provided to the typeid operator is the name of a type, the operator returns an object that identifies it. If the operand provided is an expression, typeid returns the type of the object that the expression denotes.

type_info class

This class describes the RTTI available, and is used to define the type returned by the typeid operator. This class provides to users the possibility

of shaping and extending RTTI to suit their own needs. This ability is of most interest to implementers of object I/O systems such as debuggers or database systems.

Using C++ Language Defined RTTI

To use RTTI you need to be familiar with the *dynamic cast* expression and the `typeid` operator.

The `dynamic_cast` Operator

A *dynamic cast* expression is used to cast a base class pointer to a derived class pointer. This is referred to as *downcasting*.

The `dynamic_cast` operator makes downcasting much safer than conventional static casting. It obtains a pointer to an object of a derived class that is given a pointer to a base class of that object. The operator **`dynamic_cast`** returns the pointer only if the specific derived class actually exists. If not, it returns zero.

Dynamic casts have the form:

►►—`dynamic_cast`—<—*type_name*—>—(—*expression*—)—►►

The operator converts the *expression* to the desired type *type_name*. The *type_name* can be a pointer or a reference to a class type. If the cast to *type_name* fails, the value of the *expression* is zero.

Dynamic Casts with Pointers

A dynamic cast using pointers is used to get a pointer to a derived class in order to use a detail of the derived class that is not otherwise available:

```
class employee {
public:
    virtual int salary();
};

class manager : public employee {
public:
    int salary();
    virtual int bonus();
};
```

In this class hierarchy, dynamic casts can be used to include the `manager::bonus()` function in the manager's salary calculation but not in the calculation for a regular employee. The **`dynamic_cast`** operator uses a pointer to the base class `employee`, and gets a pointer to the derived class `manager` in order to use the `bonus()` member function.

```
void payroll::calc (employee *pe) {
    // employee salary calculation
    if (manager *pm = dynamic_cast<manager*>(pe)) {
        // use manager::bonus()
    }
    else {
        // use employee's member functions
    }
}
```

If `pe` actually points to a manager object at run time, the dynamic cast is successful, `pm` is initialized to a pointer to a manager, and the `bonus()` function is used. If not, `pm` is initialized to zero and only the functions in the employee base class are used.

Note: In the above program, dynamic casts are needed only if the base class employee and its derived classes are not available to users (as in part of a library where it is undesirable to modify the source code). Otherwise, adding new virtual functions and providing derived classes with specialized definitions for those functions is a better way to solve this problem.

Dynamic Casts with References

The **dynamic_cast** operator can be used to cast to reference types. C++ reference casts are similar to pointer casts: they can be used to cast from references to base class objects to references to derived class objects.

In dynamic casts to reference types, *type_name* represents a type and *expression* represents a reference. The operator converts the *expression* to the desired type *type_name*&.

Since there is no such thing as a *zero* reference, it is not possible to verify the success of a dynamic cast using reference types by comparing the result (the reference that results from the dynamic cast) with zero. A failing dynamic cast to a reference type throws a **bad_cast** exception.

A dynamic cast with a reference is a good way to test for a coding assumption. The employee example above using reference casts is:

```
void payroll::calc (employee &re) {  
    // employee salary calculation  
    try {  
        manager &rm = dynamic_cast<manager&>(re);  
        // use manager::bonus()  
    }  
    catch (bad_cast) {  
        // use employee's member functions  
    }  
}
```

Note: This program is only intended to show the **dynamic_cast** operator used as a test. This example does not demonstrate good programming style, since it uses exceptions to control execution flow. Using **dynamic_cast** with pointers, as shown above is a better way.

The typeid Operator

The **typeid** operator identifies the exact type of an object that is given a pointer to a base class. It is typically used to gain access to information needed to perform some operation where no common interface can be assumed for every object manipulated by the system. Object I/O and database systems often need to perform services on objects where no virtual function is available to do so. The **typeid** operator enables this.

A **typeid** operation has this form:

►► typeid (—type_name—expression—) ◀◀

The result of a **typeid** operation has type `const type_info&`.

This table summarizes the results of various **typeid** operations.

Table 35. *typeid* operations

Operand	typeid Returns
<i>type_name</i>	A reference to a type_info object that represents it.
<i>expression</i>	A reference to a type_info that represents the type of the <i>expression</i> .
Reference to a polymorphic type	The type_info for the complete object referred to.
Pointer to a polymorphic type	The dynamic type of the complete object pointed to. If the pointer is zero, the typeid expression throws bad_typeid exception.
Nonpolymorphic type	The type_info that represents the static type of the <i>expression</i> . The <i>expression</i> is not evaluated.

These examples use the **typeid** operator in expressions that compare the run-time type of objects in the employee class hierarchy:

```
// ...
employee *pe = new manager;
employee& re = *pe;
// ...
typeid(pe) == typeid(employee*)    // 1. True - not a polymorphic type
typeid(&re) == typeid(employee*)  // 2. True - not a polymorphic type
typeid(*pe) == typeid(manager)    // 3. True - *pe represents a polymorphic type
typeid(re) == typeid(manager)     // 4. True - re represents the object manager

typeid(pe) == typeid(manager*)    // 5. False - static type of pe returned
typeid(pe) == typeid(employee)    // 6. False - static type of pe returned
typeid(pe) == typeid(manager)     // 7. False - static type of pe returned

typeid(*pe) == typeid(employee)   // 8. False - dynamic type of expression is manager
typeid(re) == typeid(employee)    // 9. False - re actually represents manager
typeid(&re) == typeid(manager*)   // 10. False - manager* not the static type of re
// ...
```

In comparison 1, pe is a pointer (that is, not a polymorphic type); therefore, the expression **typeid(pe)** returns the static type of pe, which is equivalent to **employee***.

Similarly, re in comparison 2 represents the address of the object referred to (that is, not a polymorphic type); therefore, the expression **typeid(re)** returns the static type of re, which is a pointer to employee.

The type returned by **typeid** represents the dynamic type of the expression only when the expression represents a polymorphic type, as in comparisons 3 and 4.

Comparisons 5, 6, and 7 are false because it is the type of the *expression* (pe) that is examined, not the type of the *object* pointed to by pe.

These examples do not directly manipulate **type_info** objects. Using the **typeid** operator with built-in types requires interaction with **type_info** objects:

```
int i;
// ...
typeid(i) == typeid(int)    // True
typeid(8) == typeid(int)    // True
// ...
```

The class **type_info** is described in “The **type_info** Class” on page 469.

The `type_info` Class

To use the `typeid` operator in Run-time Type Identification (RTTI) you must include the C++ standard header `<typeinfo.h>`. This header defines these classes:

`type_info`

Describes the RTTI available to the implementation. It is a polymorphic type that supplies comparison and collation operators, and provides a member function that returns the name of the type represented.

The copy constructor and the assignment operator for the class `type_info` are private; objects of this type cannot be copied.

`bad_cast`

Defines the type of objects thrown as exceptions to report dynamic cast expressions that have failed.

`bad_typeid`

Defines the type of objects thrown as exceptions to report a null pointer in a `typeid` expression.

Using RTTI in Constructors and Destructors

The `typeid` and `dynamic_cast` operators can be used in constructors or destructors, or in functions called from a constructor or a destructor.

If the operand of `dynamic_cast` used refers to an object under construction or destruction, `typeid` returns the `type_info` representing the class of the constructor or destructor.

If the operand of `dynamic_cast` refers to an object under construction or destruction, the object is considered to be a complete object that has the type of the constructor's or destructor's class.

The result of the `typeid` and `dynamic_cast` operations is undefined if the operand refers to an object under construction or destruction, and if the static type of the operand is not an object of the constructor's or destructor's class or one of its bases.

Understanding ILE C++ Extensions to RTTI

The ILE C++ `extended_type_info` class was designed to provide support for implementing a persistent object store. The basic operations that must be supported are:

- Allocation of memory of an object
- Allocation of memory for an array of objects
- Initial construction of an object
- Initial construction of an array of objects
- Copy construction of an object
- Copy construction of an array of objects

Additional operations for destroy and deallocation of memory are also provided to detect an exception that occurs during construction. These operations are:

- Destruction of an object
- Destruction of an array of objects
- Destruction of memory for an object

- Deallocation of memory for an array of objects

The `extended_type_info` Class

The class definitions are:

```
class extended_type_info : public type_info {
public:
    ~extended_type_info();

    virtual size_t size() const=0;

    virtual void* create(void* at) const=0; //object
    virtual void* create(void* at, size_t count) const=0; // array

    virtual void* copy (void* to, const void* from) const=0; //object
    virtual void* copy (void* to, const void* from, size_t count) const=0;
    //array

    virtual void* destroy(void* at) const=0; //object
    virtual void* destroy(void* at, size_t count) const=0; //array

    virtual void* allocObject() const=0; //object
    virtual void* allocArray(size_t count) const=0; //array

    virtual void* deallocObject(void* at) const=0; //object
    virtual void* deallocArray(void* at, size_t count) const=0; //array
};
```

Explanation of terms:

size() Size of the type represented by the `extended_type_info` object.

create(void*)

This function is called to create an object of the type represented by the `extended_type_info` object at the storage location pointed to by `at`.

create(void*, size_t)

This function is called to create an array of objects of the type represented by the `extended_type_info` object at the storage location pointed to by `at`. If any exceptions are thrown during construction, `create()` destroys the array elements that were already constructed before rethrowing the exception.

copy(void* to, const void* from)

This function is called to copy an object of the type represented by the `extended_type_info` object into the storage location pointed to by `to`, using the value of the object referred to by `from`.

copy(void* to, const void* from, size_t)

This function is called to copy an array of objects of the type represented by the `extended_type_info` object into the storage location pointed to by `to`, using the value of the object referred to by `from`. If any exceptions are thrown during construction, `copy()` destroys the array elements that were already constructed before rethrowing the exception.

destroy(void*)

This function is called to destroy an object of the type represented by the `extended_type_info` object at the storage location pointed to by `at`.

destroy(void*, size_t)

This function is called to destroy an array of objects of the type represented by the `extended_type_info` object at the storage location pointed to by `at`. If any exceptions are thrown during destruction,

`destroy()` destroys the remaining array elements that were already constructed before rethrowing the exception. If another exception is encountered during the destruction of the remaining elements, `destroy()` calls `terminate()`.

`allocObject()`

This function is called to allocate memory for an object of the type represented by the `extended_type_info` object. No initialization is performed. The user is expected to use the `create(void*)` or `copy(void*, const void*)` function to initialize the new memory.

`allocArray(size_t)`

This function is called to allocate memory for an array of objects of the type represented by the `extended_type_info` object. No initialization is performed. The user is expected to use the `create(void*, size_t)` or `copy(void*, const void*, size_t)` function to initialize the new memory.

`deallocObject(void*)`

This function is called to deallocate an object of the type represented by the `extended_type_info` object. No destruction is performed beforehand. The user is expected to use the `destroy(void*)` to terminate the object before deallocating the memory.

`deallocArray(void*, size_t)`

This function is called to deallocate an array of objects of the type represented by the `extended_type_info` object. No destruction is performed beforehand. The user is expected to use the `destroy(void*, size_t)` to terminate an array before deallocating the memory.

`linkageInfo()`

This function returns the mangled name of the class type.

Bibliography

For additional information about topics related to ILE C/400 programming, refer to the following IBM publications:

- *ADTS/400: Application Development Manager User's Guide*, SC09-2133-02, describes creating and managing projects defined for the Application Development Manager/400 feature, as well as using the program to develop applications.
- *ADTS/400: Programming Development Manager*, SC09-1771-00, provides information about using the Application Development ToolSet/400 programming development manager (PDM) to work with lists of libraries, objects, members, and user-defined options to easily do such operations as copy, delete, and rename. Contains activities and reference material to help the user learn PDM. The most commonly used operations and function keys are explained in detail using examples.
- *ADTS for AS/400: Source Entry Utility*, SC09-2605-00, provides information about using the Application Development ToolSet/400 source entry utility (SEU) to create and edit source members. The manual explains how to start and end an SEU session and how to use the many features of this full-screen text editor. The manual contains examples to help both new and experienced users accomplish various editing tasks, from the simplest line commands to using pre-defined prompts for high-level languages and data formats.
- *Application Display Programming*, SC41-5715-00, provides information about:
 - Using DDS to create and maintain displays for applications;
 - Creating and working with display files on the system;
 - Creating online help information;
 - Using UIM to define panels and dialogs for an application;
 - Using panel groups, records, or documents
- *Backup and Recovery*, SC41-5304-05, provides information about setting up and managing the following:
 - Journaling, access path protection, and commitment control
 - User auxiliary storage pools (ASPs)
 - Disk protection (device parity, mirrored, and checksum)

Provides performance information about backup media and save/restore operations. Also includes advanced backup and recovery topics, such as using save-while-active support, saving and restoring to a different release, and programming tips and techniques.

- *CICS Family: Application Programming Guide*, SC41-5454-00, provides information on application programming for CICS/400®. It includes guidance and reference information on the CICS application programming interface and system programming interface commands, and gives general information about developing new applications and migrating existing applications from other CICS platforms.
- *ILE C/C++ for AS/400 MI Library Reference*, SC09-2418-00, provides information on Machine Interface instructions available in the C for AS/400 compiler that provide system-level programming capabilities.
- *CL Programming*, SC41-5721-04, provides a wide-ranging discussion of AS/400 programming topics including a general discussion on objects and libraries, CL programming, controlling flow and communicating between programs, working

with objects in CL programs, and creating CL programs. Other topics include predefined and impromptu messages and message handling, defining and creating user-defined commands and menus, application testing, including debug mode, breakpoints, traces, and display functions.

- *Communications Management*, SC41-5406-02, provides information about work management in a communications environment, communications status, tracing and diagnosing communications problems, error handling and recovery, performance, and specific line speed and subsystem storage information.
- *Experience RPG IV Multimedia Tutorial*, SK2T-2700, is an interactive self-study program explaining the differences between RPG III and RPG IV and how to work within the new ILE environment. An accompanying workbook provides additional exercises and doubles as a reference upon completion of the tutorial. ILE RPG code examples are shipped with the tutorial and run directly on the AS/400.
- *GDDM Programming Guide*, SC41-0536-00, provides information about using OS/400 graphical data display manager (GDDM®) to write graphics application programs. Includes many example programs and information to help users understand how the product fits into data processing systems.
- *GDDM Reference*, SC41-3718-00, provides information about using OS/400 graphical data display manager (GDDM) to write graphics application programs. This manual provides detailed descriptions of all graphics routines available in GDDM. Also provides information about high-level language interfaces to GDDM.
- *ICF Programming*, SC41-5442-00, provides information needed to write application programs that use AS/400 communications and the OS/400 intersystem communications function (OS/400-ICF). Also contains information on data description specifications (DDS) keywords, system-supplied formats, return codes, file transfer support, and program examples.
- *IDDU Use*, SC41-5704-00, describes how to use the AS/400 interactive data definition utility (IDDU) to describe data dictionaries, files, and records to the system. Includes:
 - An introduction to computer file and data definition concepts
 - An introduction to the use of IDDU to describe the data used in queries and documents
 - Representative tasks related to creating, maintaining, and using data dictionaries, files, record formats, and fields
 - Advanced information about using IDDU to work with files created on other systems and information about error recovery and problem prevention.
- *ILE C/C++ Compiler Reference*, SC09-4816-00, provides reference information for the ILE C/C++ compiler. It includes compiler options, ILE C/C++ macros, preprocessor directives, and pragmas.
- *ILE C/C++ Language Reference*, SC09-4815-00, provides reference information about the ILE C/C++ compiler, including elements of the language, statements, and preprocessor directives. Examples are provided and considerations for programming are also discussed.
- *ILE C for AS/400 Run-Time Library Reference*, SC41-5607-00, provides reference information about C for AS/400 library functions, including Standard C library functions and C for AS/400 library extensions. Examples are provided and considerations for programming are also discussed.
- *ILE COBOL Programmer's Guide*, SC09-2540-02, provides information about how to write, compile, bind, run, debug, and maintain ILE COBOL programs on the AS/400 system. It provides programming information on how to call other ILE

COBOL and non-ILE COBOL programs, share data with other programs, use pointers, and handle exceptions. It also describes how to perform input/output operations on externally attached devices, database files, display files, and ICF files.

- *ILE COBOL Reference*, SC09-2539-02, provides a description of the ILE COBOL programming language. It provides information on the structure of the ILE COBOL programming language and the structure of an ILE COBOL source program. It also provides a description of all Identification Division paragraphs, Environment Division clauses, Data Division clauses, Procedure Division statements, and Compiler-Directing statements.
- *ILE COBOL Reference Summary*, SX09-1317-02, provides quick reference information on the structure of the ILE COBOL programming language and the structure of an ILE COBOL source program. It also provides syntax diagrams of all Identification Division paragraphs, Environment Division clauses, Data Division clauses, Procedure Division statements, and Compiler-Directing statements.
- *ILE Concepts*, SC41-5606-05, explains concepts and terminology pertaining to the Integrated Language Environment architecture of the OS/400 licensed program. Topics covered include creating modules, binding, running programs, debugging programs, and handling exceptions.
- *ILE RPG Programmer's Guide*, SC09-2507-03, provides information about the ILE RPG programming language, which is an implementation of the RPG IV language in the Integrated Language Environment (ILE) on the AS/400 system. It includes information on creating and running programs, with considerations for procedure calls and interlanguage programming. The guide also covers debugging and exception handling and explains how to use AS/400 files and devices in RPG programs. Appendixes include information on migration to RPG IV and sample compiler listings. It is intended for people with a basic understanding of data processing concepts and of the RPG language.
- *ILE RPG Reference*, SC09-2508-03, provides information about the ILE RPG programming language. This manual describes, position by position and keyword by keyword, the valid entries for all RPG IV specifications, and provides a detailed description of all the operation codes and built-in functions. This manual also contains information on the RPG logic cycle, arrays and tables, editing functions, and indicators.
- *ILE RPG Reference Summary*, SX09-1315-02, provides information about the RPG III and RPG IV programming language. This manual contains tables and lists for all specifications and operations in both languages. A key is provided to map RPG III specifications and operations to RPG IV specifications and operations.
- *Local Device Configuration*, SC41-5121-00, provides information about configuring local devices on the AS/400 system. This includes information on how to configure the following:
 - Local work station controllers (including twinaxial controllers)
 - Tape controllers
 - Locally attached devices (including twinaxial devices)
- *Machine Interface Functional Reference*, SC41-5810-00, describes the machine interface instruction set. Describes the functions that can be performed by each instruction and also the necessary information to code each instruction.
- *Printer Device Programming*, SC41-5713-04, provides information to help you understand and control printing. Provides specific information on printing elements and concepts of the AS/400 system, printer file and print spooling support for printing operations, and printer connectivity. Includes considerations for using personal computers, other printing functions such as Business Graphics

Utility (BGU), advanced function printing (AFP), and examples of working with the AS/400 system printing elements such as how to move spooled output files from one output queue to a different output queue. Also includes an appendix of control language (CL) commands used to manage printing workload. Fonts available for use with the AS/400 system are also provided. Font substitution tables provide a cross-reference of substituted fonts if attached printers do not support application-specified fonts.

- *REXX/400 Programmer's Guide*, SC41-5728-00, provides a wide-ranging discussion of programming with REXX on the iSeries system. Its primary purpose is to provide useful programming information and examples to those who are new to Procedures Language 400/REXX and to provide those who have used REXX in other computing environments with information about the Procedures Language 400/REXX implementation.
- *ILE RPG Programmer's Guide*, SC09-2507-03, provides information needed to design, code, compile, and test RPG programs on the iSeries system. The manual provides information on data structures, data formats, file processing, multiple file processing, the automatic report function, RPG command statements, testing and debugging functions, application design techniques, problem analysis, and compiler service information. The differences between the RPG for AS/400 compiler, the System/38[®] environment RPG III compiler, and the System/36[®]-compatible RPG II compiler are also discussed.
- *iSeries Security Reference*, SC41-5302-04, tells how system security support can be used to protect the system and the data from being used by people who do not have the proper authorization, protect the data from intentional or unintentional damage or destruction, keep security information up-to-date, and set up security on the system.
- *Local Device Configuration*, SC41-5121-00, provides step-by-step procedures for initial installation, installing licensed programs, program temporary fixes (PTFs), and secondary languages from IBM. This manual is also for users who already have an AS/400 system with an installed release and want to install a new release.
- *System API Programming*, SC41-5800-00, provides information for the experienced application and system programmers who want to use the OS/400 application programming interfaces (APIs). Provides getting started and examples to help the programmer use APIs.
- *System Operation*, SC41-4203-00, provides information about handling messages, working with jobs and printer output, devices communications, working with support functions, cleaning up your system, and so on.

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area.

Any reference to an IBM product, program, or service is not intended to state or imply that only IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

Director of Licensing,
Intellectual Property & Licensing
International Business Machines Corporation,
North Castle Drive, MD - NC119
Armonk, New York 10504-1785,
U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independent created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Canada Ltd.
Department 071
1150 Eglinton Avenue East

Toronto, Ontario M3C 1H7
Canada

Such information may be available, subject to appropriate terms and conditions, including in some cases payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

This publication contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

Programming Interface Information

This edition applies to Version 5, Release 1, Modification 0, of IBM WebSphere Development Studio (5722-WDS), ILE C/C++ compilers.

This book is intended to help you create Integrated Language Environment C/C++ programs. It contains information necessary to use the Integrated Language Environment C/C++ compiler. This book documents general-use programming interfaces and associated guidance information provided by the Integrated Language Environment C/C++ compiler.

Trademarks and Service Marks

The following terms are trademarks of the International Business Machines Corporation in the United States or other countries or both:

400	IBM
AFP	IBMLink
AS/400	Integrated Language Environment
AS/400e	iSeries
Application System/400	OS/2
C/400	OS/400
CICS/400	RPG/400
COBOL/400	SAA
DB2	SQL/400
@server	WebSphere
GDDM	

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.

Industry Standards

The Integrated Language Environment C compiler and run-time library are designed according to the ANSI for C Programming Languages - C ANSI/ISO 9899-1990 standard.

The C++ language is consistent with the International Standard for Information Systems - Programming Language C++ - ISO/IEC 14882:1998 standard.

Index

A

- activation 30, 70
- activation groups
 - grouping 71
 - leaving 56
 - process 70
 - reasons to use 30
- adding
 - conditional breakpoints 97
 - programs to a debug session 93
 - service programs to a debug session 93
- additional types of exception handlers 257
- argument passing
 - match data type requirements 346
 - operational descriptors 346, 349
- attribute of an ILE C program 23

B

- before starting debug 87
- binary stream database files
 - I/O considerations 207
- binary stream display files
 - program devices 221
- binary stream ICF files
 - I/O considerations 236
 - program devices 236
- binary stream save files
 - I/O considerations 253
- binary stream subfiles
 - I/O considerations 221
- binary stream tape files
 - I/O considerations 246
- binary streams 143
- bindable api
 - CEEHDLR 281
 - CEEHDLU 281
- binder 26
- binder language
 - EXPORT symbol 65
 - reason to use 33
 - STRPGMEXP
 - LVLCHK parameter 65
- binding directory
 - reasons for creating 26
- block records 429
- breakpoints
 - conditional 96
 - remove at a statement number 98
 - setting 96
 - unconditional 96
 - unconditional, setting and removing 99

C

- C locale migration table 412

- CALL command
 - changes to parameters 76
 - passing parameters to a program 75, 76
- calling
 - C++ procedures 351
 - C++ programs 351
 - CL 328
 - conventions for dynamic program calls 326
 - ILE C 343
 - message queue 257
 - OPM COBOL 328
 - OPM RPG 328
 - procedures 345, 346
- cancel handlers, using 278
- CCSIDs recognized single-byte EBCDIC CCSID 403
- CEEMRCR 285
- changing
 - CCSID 405
 - module 95
 - module optimization level 124
 - observability 123
 - optimization 123
 - optimization levels 123
 - value of scalar variables, while debugging 113
 - value of variables 111
 - view of a module 96
- character arrays, displaying 117
- Character Data Representation Architecture (CDRA) 403
- character sets 411
- characters
 - case conversion 411
 - classification 411
 - collating 411
 - ordering 411
- checking
 - errno value 260
 - global variable _EXCP_MSGID 261
 - Machine Instruction template 124
 - major/minor return code 261
 - optimization level 124
 - return value of a function 259
 - system exceptions for record files 261
 - system exceptions for stream files 261
- CL, calling 328
- classes, exception 269
- Coded Character Set Identifier (CCSID)
 - changing 405
 - Character Data Representation Architecture (CDRA) 403
 - character set 403
 - code page 403
 - code points 403
 - definition 403
 - graphic characters 403

- Coded Character Set Identifier (CCSID)
 - (*continued*)
 - specifying 404
- coded character sets 411
- coding procedures and data items 65
- command
 - ACQPGMDEV parameter 237
 - Add ICF Device (ADDICFDEVE) 237
 - break 109
 - Change ICF File (CHGICFF) 237
 - Change Module (CHGMOD) 125
 - Create a Display File (CRTDSPF) 221
 - Create Binding Directory (CRTBNDDIR) 26
 - Create Bound C Program 22
 - Create Bound C Program (CRTBNDC) 22, 90
 - Create Bound C++ Program (CRTBNDCPP) 22
 - Create C Module (CRTCMOD) 24, 90
 - Create C++ Module (CRTCPMOD) 24
 - Create DDM file (CRTDDMF) 207
 - Create ICF File (CRTICFF) 237
 - Create Program (CRTPGM) 22, 67
 - ACTGRP(*NEW) 67
 - BNDDIR parameter 66
 - default parameters 22
 - ENTMOD parameter 67
 - OPTION(*DUPPROC) 67
 - OPTION(*DUPVAR) 67
 - OPTION(*RSLVREF) 67
 - Create Service Program (CRTSRVPGM) 65, 66
 - ACTGRP(*CALLER) 66
 - EXPORT(*ALL) 65
 - EXPORT(*SRCFILE) 65
 - OPTION(*DUPPROC) 66
 - OPTION(*RSLVREF) 66
 - debug 88
 - End Debug (ENDDBG) 90
 - End Program Export (ENDPGMEXP) 65
 - Override Diskette File (OVRDKTF) 249
 - Override ICF Device (OVRICFDEVE) 237
 - Override ICF File (OVRICFF) 237
 - Start Debug (STRDBG) 90, 94
 - Start Program Export (STRPGMEXP) 65
 - Update Production files (UPDPROD) 94
 - Work with Module (WRKMOD) 124
- command line, debug 88
- common mechanism to return function results 327
- compile-time errors 320
- condition handler
 - move the resume cursor 285
 - percolate an exception 283

- condition handler (*continued*)
 - promote an exception 285
 - register 281
 - resume cursor, move 285
- condition token 285
- conditional breakpoints
 - adding 97
 - set to a statement 98
 - setting, example 100
- constructs, displaying, sample
 - source 129
- control boundary 267
- control the public interface 65
- conventions, calling 326
- converting
 - *CLD objects types to *LOCALE object types 412
 - from packed decimal data types 309
 - packed decimal type to floating point type 312
 - packed decimal type to integer type 311
 - packed decimal type to packed decimal type 309
 - string literals in a source file 405
- coordinating listing and debug view
 - options 88
- CPYTOSTMF command 172
- creating
 - binding directory 26
 - debug views 92
 - include source view 91
 - listing view 88, 91
 - locales 416
 - program 19
 - program in one step
 - compiling and binding 22
 - program in two steps
 - compiling and binding 24
 - root source view 90
 - service program 26
 - source file 404
 - source physical file with a CCSID 404
 - temporary module 22
 - view 90

D

- data description specification 202
- data type compatibility 333, 341
- database files
 - access path 203
 - arranging key fields 203
 - arrival sequence access path 203
 - binary stream functions 207
 - commitment control 213
 - comparison with stream file 168
 - data file 202
 - definition 201
 - duplicate key values 203
 - field level description 201
 - input and output fields 190
 - key fields 190
 - keyed sequence access path 203
 - null capable fields 205
 - open as record files 206

- database files (*continued*)
 - opening as binary stream files 206
 - record at a time processing 207
 - record functions 206
 - record level description 201
 - source member 202
- database record
 - arrival sequence 207
 - delete 204
 - keyed sequence 209
 - lock conditions 204
 - record I/O functions 210
- DBGVIEW
 - *ALL 90, 91
 - *EXPMAC 91
 - *LIST 91
 - *NONE 91
 - *SHOWINC 91
 - *SHOWSKP 91
 - *SHOWSYS 91
 - *SHOWUSR 91
 - *SOURCE 90, 91
- DDM files
 - binary stream functions 207
 - definition 201
 - I/O considerations 206, 207
 - open as binary files 206
 - open as record files 206
 - opening as binary stream files 206
 - record functions 206
 - sharing 204
- debug command
 - clear program 97
 - display module 95, 96
 - equate 114
 - eval 111
 - qual 111
 - step 109
 - step into 110
 - STEP INTO 106
 - step over 109
 - STEP OVER 106
 - using the eval debug command 112
- debug command line 88
- debug data
 - creating 88
 - definition 88
 - effect on object size 88
 - none 91
- debug options
 - example, setting 94
 - setting 94
- debug session
 - adding programs 93
 - adding service programs 93
 - removing programs 93
 - removing programs from debug session 93
 - removing service programs 93
 - removing service programs from debug session 93
 - starting 90, 92
 - starting for OPM programs 92
- debug view
 - definition 88
 - preparing a program for debugging 88

- debug views, creating 92
- debugging a program
 - limitations of debug expression grammar 87
 - preparing a program 88
- declaring pointer variables 300, 301
- default program device
 - acquiring 226
 - changing 228
- default program type
 - *PGM 93
 - *SRVPGM 93
- determine the value of an expression 112
- device files
 - both fields 193
 - indicator field 194
 - input fields 192
 - OS/400 feedback areas 219
 - output fields 193
 - separate indicator area 194, 196
 - INDARA keyword 194
 - part of the file buffer 196
- different passing methods 345
- direct monitor handlers
 - using 268
- diskette files
 - binary stream functions 250
 - blocking 250
 - I/O considerations 249, 250
 - open as binary stream files 249
 - open as record files 250
 - record functions 250
- display files
 - binary stream functions 221
 - changing default program device 222
 - definition 220
 - I/O considerations 220
 - indicators 219
 - major/minor return codes 220
 - open as record files 221
 - opening as binary stream files 221
 - record functions 222
 - separate indicator areas 219
 - subfiles 220
- displaying
 - character arrays 117
 - characters to a newline 117
 - constructs, sample source 129
 - null-ended character arrays 116
 - structure 115
 - system and space pointers, sample source 127
 - templates, while debugging 122
 - value of variables 111
 - variables as hexadecimal values 115
- DSPMODSRC 92
- dynamic program call 326

E

- elements of a language environment 411
- entering source statements 21
- environment variables
 - locale 418
- equating a name with a variable expression or command 114

- errno value, checking 260
- error macros
 - EIOERROR 261
 - EIORECERR 261
- errors
 - compile-time 320
 - packed decimal data type 320
 - run-time 320
- EVAL debug command
 - sample source 126
- examples
 - creating a program in one step 23
 - creating a program in two steps 55
 - diskette files, writing to 250
 - entering source statements 20
 - opening text stream files 147
 - packed decimal type to floating point type 312
 - packed decimal type to integer type 311
 - packed decimal type to packed decimal type 310
 - pointer casting 304
 - pointer declaration 301
 - pragma mapinc 190
 - setting debug options 94
 - tape file, writing to 247
 - unconditional breakpoints 97
 - updating a service program 50
 - using the `_Racquire()` function 226
- exception classes 269
- exception handlers
 - additional types 257
 - direct monitor handlers, using 265
 - HLL-specific handlers, using 265
 - ILE condition handlers, using 265
 - priority 257
- exception handling mechanism 281
- exception message
 - list of 258
- exceptions
 - example 271
 - nested 278
 - unhandled 267
- export
 - data item 64
- externally described device files, using 192
- externally described files 201, 202
 - definition 185
 - field level descriptions 202
 - logical database files 190
 - mapinc 185
 - physical database files 190
 - record format name 188
 - using 190

F

- feedback information 428
- file
 - database 201
 - description 145
 - logical 202
 - naming conventions 146
 - objects 145
 - open 430

- file (*continued*)
 - physical 201, 431
 - shared 430
- format
 - date 411
 - fopen 142, 170
 - monetary quantities 411
 - numbers 411
 - SYSIFCOPT 167
 - system responses 411
 - time 411
- functions
 - `_GetExcData` 287
 - `_Racquire` 222, 226
 - `_Rdevatr` 219
 - `_Rformat` 222
 - `_Rindara` 219, 226
 - `_Rpgmdev` 222
 - `_Rpoen` 222
 - `_Rreadindv` 222
 - `_Rreadnc` 222
 - `_Rrelease` 222
 - fopen 143, 221
 - locale-sensitive run-time 419
 - lrecl parameter 143, 249
 - perror 260
 - raise 287
 - record I/O 427
 - return value 259
 - signal 287, 288
 - signals raised 288
 - stepping into 110
 - stepping over 109
 - stream 431
 - strerror 260

G

- global variables
 - `_EXCP_MSGID` 261

H

- handle cursor 258
- handle errors 262
- handle the signal 288
- header file
 - `<decimal.h>` 309
 - `<errno.h>` 260
 - `<except.h>` 270
 - `<recio.h>` 205
 - `<signal.h>` 288
 - `<stdio.h>` 142
- HLL-specific handlers
 - example 289
 - using 287

I

- I/O considerations
 - binary stream database files 207
 - binary stream ICF files 236
 - binary stream save files 253
 - binary stream subfiles 221
 - binary stream tape files 246
 - DDM files 207

- I/O considerations (*continued*)
 - record diskette files 250
 - record display files 219
 - record subfiles 221
- ICF files
 - binary stream functions 236
 - change the default program device 237
 - I/O considerations 236
 - indicators 219
 - major/minor return codes 220
 - open as binary stream files 236
 - open as record files 237
 - record functions 238
 - separate indicator areas 219
- ILE C program attributes 23
- ILE C, calling 343
- import
 - data item 61
 - definition 26
- include source view, creating 91
- indicators
 - as part of the file buffer, specifying 223
 - definition 219
 - INDARA 219
 - option indicators 219
 - response indicators 219
 - separate indicator areas 219
 - types 219
- indicators as part of the file buffer
 - INDARA keyword 224
- indicators in a separate indicator area 224
- integrated file system 167
- Integrated Language Environment
 - bindable API 281, 285
- Integrated Language Environment
 - condition handler 281
- Integrated Language Environment
 - condition handlers 281, 283, 285
- international locale support
 - ILE C support 411

L

- language environment 411
- LC_ALL locale variable 417
- LC_COLLATE locale variable 417
- LC_CTYPE locale variable 417
- LC_MONETARY locale variable 417
- LC_NUMERIC locale variable 417
- LC_TIME locale variable 417
- LC_TOD locale variable 417
- library names 432
- limitations of debug expression grammar 87
- limits of source debugger 87
- listing 91
- listing view, creating 88, 91
- locale definition
 - POSIX 415, 419
 - SAA 419
- locales
 - *CLD object types 412
 - *LOCALE object types 412
 - customizing 417

- locales (*continued*)
 - environment variables 418
 - ILE C for AS/400 support 412
 - international locale support 411
 - library function 417
 - overview of ILE C support 411
 - run-time functions 419
 - setting an active locale 417
- LOCALETYPE option 416
- locate mode 429
- locate run-time errors 87
- logical files
 - multi-format 202

M

- major/minor return code, checking 261
- major/minor return codes 220
- match data type requirements
 - by reference 345
 - by value directly 345
 - by value indirectly 345
- module
 - changing 95
 - changing the view 96
 - different views 96
 - effect of debug data on size 88
 - observability 125
 - removing 125
 - preparing for debugging 88
- module object
 - debug data 24
 - program entry procedure 24
 - user entry procedure 24
- module's optimization level,
 - changing 124
- multiple record formats, using 196

N

- native language 411
- nested exceptions 278
- newline, displaying 117
- no debug data 91
- null capable fields 205
- null ended character arrays,
 - displaying 116

O

- observability 125
- observability, changing 123
- open data path 204
- open files 430
- open modes for dynamically created
 - stream files 143
- opening
 - database files as binary stream
 - files 206
 - DDM files as binary stream files 206
 - display files as binary stream
 - files 221
 - display files as record files 221
 - subfiles as binary stream files 221
- opening binary stream files
 - character at a time 150

- opening binary stream files (*continued*)
 - record at a time 156
- opening text stream files
 - fopen 147
 - modes 147
- OPM COBOL, calling 328
- OPM default activation groups 56
- OPM RPG, calling 328
- optimization level
 - *FULL 124
 - *NONE 124
 - changing 123
- optimization, changing 123
- optimizing 123
- OPTION
 - *EXPMAC 91
 - *LSTDBG 91
 - *SHOWINC 91
 - *SHOWSKP 91
 - *SHOWSYS 91
 - *SHOWUSR 91
 - *SRCDBG 91
 - *XREF 64
- overflow behavior 313

P

- packed decimal data
 - conversion functions 200
- packed decimal data type
 - representation 309
- packed decimal types
 - pragma nosigntrunc directive 322
- passing
 - arguments 345
 - different methods 345
 - packed decimal arguments 315
 - packed decimal value to a
 - function 313
 - pointer to a packed decimal to a
 - function 314
 - pointers as arguments 304
 - styles 345
- percolation 258
- physical file 431
- physical files 201
- pointers
 - casting 303
 - casting constraints 303
 - constraints 300
 - declaring pointer variables 300, 301
 - function 299, 301
 - invocation 299
 - label 299
 - open 299, 300
 - passing pointers 304
 - pointers other than open
 - pointers 300
 - space 299
 - suspend 299
 - system 299
 - teraspaces 395
 - types 299
- pragma
 - argument 345
 - convert 403, 405
 - disable_handler 268

- pragma (*continued*)
 - exception_handler 268, 269, 273
 - externally described files 185
- pragma mapinc
 - database files 190
 - header description 186
 - lvlchk option 187
 - typedefs 185
- pragma mapinc directive 63
- preparing a program for debugging 88
- printer files
 - binary stream functions 243
 - FCFC 242
 - I/O considerations 219, 243
 - indicators 219
 - major/minor return codes 220
 - open as binary stream files 243
 - open as record files 243
 - record functions 243
 - separate indicator areas 219
- procedure pointer calls 346
- procedures
 - calling 345, 346
 - stepping into 110
 - stepping over 109
- program
 - debugging 87
 - devices
 - I/O feedback area 231
 - effect of debug data on size 88
 - preparing for debugging 88
 - stepping into 106
 - stepping over 106
 - stepping through 106
- program described files 201
- program entry procedure (PEP) 24
- program source, viewing 95
- public interface 33, 65

Q

- QCAPCMD 76
- QINLINE 145
- QUSRTOOL
 - MAKE 20

R

- reading binary stream files
 - character at a time 152
 - record at a time 157
- reading text stream files 149
- record diskette files
 - blocking 250
 - I/O considerations 250
 - read and write 250
- record display files
 - I/O considerations 222
- record field names 189
- record files 141
- record format
 - _Rformat() function 188
 - definition 188
- record ICF files
 - I/O considerations 237
 - program devices 237

- record save files
 - I/O considerations 253
- record subfiles
 - I/O considerations 222
- record tape files
 - blocking 247
 - I/O considerations 246
 - using _Rfeod 246
 - using _Rfeov 247
- removing
 - breakpoints 96
 - breakpoints at a statement number 98
 - module observability 125
- removing all breakpoints 101
- removing breakpoints
 - all 101
 - unconditional breakpoints 99
- reserving storage 189
- resume point 258
- retrieving
 - return value 343
- return function results 327, 343
- return value of a function, checking 259
- RIOFB_T 428
- root source 90
- root source view, creating 90
- run-time
 - run-time errors 320, 322
 - suppress run-time error 322
- run-time model 69
- ANSI C semantics 69

S

- save files
 - binary stream functions 253
 - I/O considerations 252
 - open as binary stream files 252
 - open as record files 253
 - record functions 253
- scalar variables
 - arrays 111
 - structures 111
- scalar variables, changing value while debugging 113
- service program
 - public interface 33
 - reasons for creating 26, 33
- service programs 33
- session manager 145
- setting
 - active locale 417
 - breakpoints 96
 - conditional breakpoint to a statement 98
 - debug options 94
- setting breakpoints
 - unconditional breakpoints 99
- shared file 430
- SIG_DFL 288
- SIG_IGN 288
- signal
 - SIGABRT 288
 - SIGFPE 288
 - SIGILL 288
 - SIGINT 288

- signal (*continued*)
 - SIGIO 288
 - SIGOTHER 288
 - SIGSEGV 288
 - SIGTERM 288
 - SIGUSR1 288
 - SIGUSR2 288
- signal handler
 - change the state of nested 267
- signal handling 288
- signature 33, 65
- source file conversion 404
- source stream file (SRCSTMF) 172
- space pointers, displaying, sample source 127
- specifying CCSID 404
- SRCSTMF parameter 172
- starting
 - debug session 92
 - OPM debug session 92
 - source debug session 90
- static procedure call 345
- STEP debug command
 - into 106
 - into, example 107
 - over 106
- step function 106
- stepping
 - into a program 106
 - into a program, example 107
 - into procedures 110
 - over a program 106
 - over procedures 109
 - through program 106
- STRDBG 92
- stream buffering
 - fully buffered 142
 - line buffered 142
 - unbuffered 142
- stream files 141, 142, 167
- structure, displaying 115
- subfiles
 - binary stream functions 221
 - definition 220
 - I/O considerations 221
 - opening as binary stream files 221
 - record functions 222
 - using 233
- suspend point 258
- system buffer 429
- system exceptions
 - record files, checking 261
 - stream files, checking 261
- system pointers, displaying sample source 127

T

- tape files
 - binary stream functions 246
 - blocking 246
 - I/O considerations 245
 - open as binary stream files 246
 - open as record files 246
 - processing 430
 - record functions 247
- templates, displaying 122

- teraspaces 395
 - 16-byte pointers 395
 - 8-byte pointers 395
 - pointers 395
 - process local pointers 395
- text streams
 - definition 143
 - stderr 144
 - stdin 144
 - stdout 144
- trigraphs 22
- types of handlers 265

U

- unconditional breakpoints
 - removing 96
 - setting 96
 - setting and removing 99
- understanding packed decimal data type errors 320
- unhandled exceptions 267
- unicode 406
- updating
 - binary stream files 153
 - service program 50
 - text stream files 150
- user entry procedure (UEP) 24
- using
 - CALL command 75
 - cancel handlers 278
 - CRTSQLCI command 23
 - direct monitor handlers 268
 - environment variables 418
 - eval debug command 112
 - externally described device files 192
 - externally described logical database files 190
 - externally described physical database files 190
 - Integrated Language Environment
 - condition handlers 281
 - library functions with a packed decimal data type 316
 - multiple record formats 196
 - packed decimal data 199
 - packed decimal data type 309
 - trigraphs in place of C characters 22
 - zoned decimal data 199

V

- value of scalar variables, changing while debugging 113
- value of variable
 - changing 111
 - displaying 111
- variable, expression or command, equating a name 114
- variables
 - displaying as hexadecimal values 115
- view
 - creating 90
 - different module views 96
 - include source 91

view (*continued*)
 listing 91
 program source 95
 root source 90

W

writing binary stream files
 character at a time 152
 record at a time 156
writing text stream files 148

Z

zoned decimal data
 using 199



Program Number: 5722-WDS

SC09-2712-03

